DESIGN OF A MULTICHANNEL OUTDOOR DATA LOGGER FOR PRECISE TEMPERATURE MEASUREMENTS

By

Kirill Dolgikh

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

In

Electrical Engineering

University of Alaska Fairbanks

December 2018

APPROVED:

Dr. Dejan Raskovic, Committee Chair
Dr. Vladimir Romanovsky, Committee Member
Dr. Denise Thorsen, Committee Member
Dr. Charles E Mayer, Chair
Department of Electrical and Computer Engineering
Dr. Douglas Goering, Dean
College of Engineering and Mines
Dr. Michael Castellini, *Dean of the Graduate School*

Abstract

In this thesis, we present the prototype of a multichannel data logger (the Logger) for precise temperature measurements. Its intended application is to take soil (permafrost) temperature measurements with a thermistor probe or thermistor string. However, its hardware and firmware architectures are quite flexible, so it can be used in other applications. The Logger has 16 channels. In addition to the internal memory, it supports microSD-cards of up to 2 GB, which allows it to store up to 41.9 million measurements for each channel. The Logger's estimated battery life is 8 years when making measurements once per hour. It has a radio transmitter, which will allow it to download data wirelessly and potentially participate in a wireless sensor network once the appropriate firmware is developed. Currently, only the communication protocol with the radio is implemented, while the radio-to-radio protocol is under development. The Logger is small - only 6 x 1 x 1 inches and the final product will be even smaller. Components are rated down to -40° C and the Logger successfully passed testing at -30° C. After the extensive testing to ensure performance it has been shown that the Logger outperforms the Campbell Scientific, Inc. CR1000 logger and exceeds the design requirements. Measured temperature resolution of the Logger is below 2.5 mK in the entire temperature range. The Logger's equivalent temperature accuracy, which was determined using a known resistive input, is below 10 mK within -25° C to 40° C and below 20 mK elsewhere. The developed calibration technique provides the equivalent accuracy below 0.3 mK within -40° C to 40° C. To provide an accuracy of ±0.01° C when making temperature measurements with thermistors, the Logger should be calibrated against a thermometer that has been calibrated as a secondary standard, which will be done in the future.

Table of Contents

Page

Abstract iii

Table of Contents iv

List of Figures x

List of Tables xiii

Acknowledgements xiv

Chapter 1 Introduction 1

1.1 Background 1

1.2 Overview of Commercially Available Loggers 3

1.3 Objectives 3

1.4 Overview of the Designed Logger 5

1.5 Overview of the Results 5

1.6 Competitive Advantages of the Logger 6

1.7 Comparison to Similar Devices Developed in Academia 7

1.8 Thesis Organization 8

Chapter 2 Hardware Architecture 10

2.1 Design of a Temperature Acquisition Module 11

2.1.1 Schemes for a Sensing Circuit 11

2.1.2 The Tentative Study of the Schemes for the Sensing Circuit 12

2.1.3 The ADC Selection 19

2.1.4 The Sensing Scheme Selection 22

2.1.5 The Modification of the Logger's Sensing Circuit 24

2.1.6 Studying the VE Configuration #3 27

2.2 Design of the Power Supply Module 30

2.2.1 Selecting the Power Supply Components 30

2.2.2 Calculating the Expected Battery Life 34
2.2.3 The Voltage Regulator Efficiency and Power Dissipation 40
2.2.4 Other Supporting Hardware 41
2.3 PCB Design and the Final Product 42
Chapter 3 Firmware Architecture 44
3.1 Main program 44
3.1.1 Initialization 47
3.1.2 Functions to Measure Resistance 48
3.1.3 The USB Functions 50
3.2 The Peripherals' Drivers 52
3.2.1 Driver for the ADG706 Multiplexer 52
3.2.2 Driver for the ADS1247 ADC 52
3.2.3 SD-card 54
Chapter 4 User Interface 55
4.1 General Information on the User Interface 55
4.2 Get Conversion Result 57
4.2.1 Converting the Measured Resistance into Temperature 58
4.3 Functions to Get the Logger Parameters 60
4.4 Functions to Set the Logger Parameters 60
Chapter 5 Experimental Results 62
5.1 General Information on Experiments 62
5.2 Equivalent Temperature Accuracy with Resistive Input 65
5.2.1 Measurements at Room Temperature 65
5.2.2 Measurements at Low Temperatures 85
5.3 The Ice-Bath Measurement 88
5.4 Air temperature measurements 90

5.5 Supply Current Measurements 92
Chapter 6 Conclusion and Future Work 96
References 98
Appendix A 103
The Logger's Schematic. VE Scheme Configuration #2. 103
Appendix B 105
The Logger's Printed Circuit Board 105
Appendix C 110
The Test Bed with Resistors 110
Appendix D 112
The ADS1247 PGA Simulation 112
Appendix E 118
The Experiment's Data for the Logger 118
Appendix F 121
The Microcontroller's Firmware 121
main.c 121
init.c 125
init.h 127
adc_driver.c 128
adc_driver.h 140
crc8.c 149
crc8.h 151
data_prep.c 152
data_prep.h 153
hal_SPI.c 154
hal_SPI.h 156

measure.c 156
measure.h 161
mux_adg706_driver.c 162
mux_adg706_driver.h 165
real_time_clock.c 167
real_time_clock.h 168
sdcard.c 169
sdcard.h 178
send_over_usb.c 181
send_over_usb.h 189
SPI_Library.c 191
SPI_Library.h 195
SPI_Pins.h 197
usbEventHandling.c 198
sleep_timer.c 202
sleep_timer.h 203
states.h 204
ucs_functions.c 205
ucs_functions.h 207
Appendix G 209
The User Interface Code 209
cli.m 209
request_num_bytes.m 212
num_bytes_RXed_callback.m 213
data_received_callback.m 214
ProcessInputData.m 214

transform_meas_rate.m 218
transform_conv_result.m 218
PresentConvResult.m 218
ProcessConvResult.m 221
analyse_logger_data.m 224
analyse_logger_data_all_ch.m 224
ProcessInputBulkData.m 225
get_bugger_calback.m 227
get_adc_param.m 227
param_received_callback.m 228
get_measurement_rate.m 231
meas_rate_received_callback.m 231
get_adc_cal_register.m 231
cal_reg_received_callback.m 232
get_rt_clock.m 232
clock_received_callback.m 233
set_measurement_rate.m 233
send_set_cmd.m 236
ack_received_callback.m 237
set_measure_start.m 237
set_adc_data_rate.m 238
set_num_of_bulk_segments.m 239
set_adc_vrefcon_reg.m 240
set_sysocal_enable.m 240
set_delay_enable.m 241
set_rt_clock.m 241

set_keep_data_in_memory.m 242
set_cal_enable.m 242
set_operational_mode.m 242
cli_help.m 243
cli_help_vrefcon.m 244

List of Figures

Page

Figure 2-1. The Logger's hardware block-scheme 10
Figure 2-2. A thermistor probe schematic diagram 11
Figure 2-3. The PS103J2 10 kΩ thermistor resistance versus temperature 12
Figure 2-4. Four configurations of the voltage excitation (VE) scheme 13
Figure 2-5. The thermistor's dissipated power with Rref = 100 kΩ 14
Figure 2-6. VE configuration #1: ADC input voltage (left) and voltage resolution required for 0.01° C temperature resolution (right) 15
Figure 2-7. A simple bridge scheme 16
Figure 2-8. Bridge scheme: ADC input voltage (left) and voltage resolution required for 0.01° C temperature resolution (right) 16
Figure 2-9. A simple current excitation (CE) scheme 17
Figure 2-10. CE scheme: the ADC voltages 18
Figure 2-11. Voltage resolution of the CE and the VE schemes 19
Figure 2-12. Common-mode voltages for the VE configurations 1 – 4 26
Figure 2-13. The RT-curves for 5 kΩ, 10 kΩ, and 20 kΩ thermistors (left) and the minimum voltage resolution for 0.01° C temperature resolution for 10 kΩ and 20 kΩ thermistors (right) 29
Figure 2-14. Expected temperature resolution for 10 kΩ (left) and 20 kΩ (right) thermistors for all data rates 29
Figure 2-15. Expected temperature resolution for 10 kΩ (left) and 20 kΩ (right) thermistors for 5 SPS and 10 SPS 30
Figure 2-16. The Energizer L91 discharge profile (left) and temperature effects on capacity (right). Taken from [35] 33
Figure 2-17. Discharge curves for Panasonic CR-2 battery. Taken from [38]. 34
Figure 2-18. Discharge curves (left) and temperature effects on capacity (right) for Hitachi-Maxell CR-2 battery. Taken from [39] 34
Figure 2-19. One cycle of the Logger's operation 35
Figure 2-20. Average current (left) and battery life (right) for the minimal measurement interval 40
Figure 2-21. Average current (left) and battery life (right) for 1-hour measurement interval 40
Figure 2-22. The prototype of the Logger (Board 2) 43
Figure 3-1. The firmware block diagram 44

Figure 3-2. The main program's flowchart 45
Figure 3-3. Data segment structure 48
Figure 3-4. Measuring in the "RTC mode" flowchart 49
Figure 3-5. Send conversion result function 51
Figure 3-6. ADC SPI commands 52
Figure 4-1. The S-H equation error in the range of -55° C to 76° C (left) and 0° C to 76° C (right) 58
Figure 4-2. The S-H equation error for operating points -40°C, 0°C, 40°C (left) and -20°C, 0°C, 20°C (right) 59
Figure 4-3. The difference in adjacent resistances in the range -1.15° C to -0.95° C for the PS103J2 thermistor 59
Figure 5-1. Resistive test bed 63
Figure 5-2. The CR1000 test bed 64
Figure 5-3. 8160 samples for the 10.7 kΩ resistor. Board 2, DR = 10 SPS 66
Figure 5-4. Drift of the 16 kΩ (left) and the 147 kΩ (right) resistors. 8160 and 5100 samples respectively. Board 2, DR = 10 SPS 67
Figure 5-5. Outliers at the session start for the 24.5 kΩ (left) and the 109 kΩ (right) resistors. Board 2, DR = 10 SPS, 8160 samples 67
Figure 5-6. Difference in average measured resistance in Ω (left) and % (right) and percent difference between the two boards 71
Figure 5-7. Difference between resistances measured with the Logger at 10 SPS and the reference ohmmeter 71
Figure 5-8. Board 2: the instantaneous and the average temperature accuracy for 10 kΩ (left) and 20 kΩ (right) thermistors at 10 SPS 73
Figure 5-9. Board 2: the average accuracies for 10 kΩ and 20 kΩ thermistors 74
Figure 5-10. Board 2: the expected and the measured temperature resolutions for 10 kΩ and 20 kΩ thermistors at 10 SPS 74
Figure 5-11. Board 1: the instantaneous and the average temperature accuracy for 10 kΩ (left) and 20 kΩ (right) thermistors at 5 SPS 75
Figure 5-12. Board 1: comparison of the average temperature accuracies for 10 kΩ and 20 kΩ thermistors at 5 SPS 76
Figure 5-13. Board 1: expected and measured temperature resolutions for 10 kΩ and 20 kΩ thermistors at 5 SPS 76

Figure 5-14. The equivalent uncalibrated and calibrated temperature accuracies for the 20 kΩ thermistor 77
Figure 5-15. Measured resistance values and their distribution for the 147 kΩ resistor 79
Figure 5-16. Measured resistances for the 665 kΩ and the 487 kΩ resistors 80
Figure 5-17. The CR1000 equivalent temperature accuracy for 10 kΩ (left) and 20 kΩ (right) thermistors 81
Figure 5-18. The minimum voltage resolution required for a 0.01 °C temperature resolution (left) and the expected temperature resolution (right) for 10 kΩ and 20 kΩ thermistors 82
Figure 5-19. The resistance accuracy for the Logger and the CR1000 83
Figure 5-20. The uncalibrated equivalent temperature accuracy for the Logger and the CR1000: 10 kΩ (left) and 20 kΩ (right) thermistors 84
Figure 5-21. The example of a low temperature measurement 87
Figure 5-22. The ice-bath temperature measurement 88
Figure 5-23. The thermistor temperature during equilibrium with media 89
Figure 5-24. The test setup for outside measurements 90
Figure 5-25. Air temperature measurement on Feb 8, 2018: 5-hours interval (left) and 50-minutes interval (right) 91
Figure 5-26. Air temperature measurement on Feb 8, 2018: 3-minute interval for the Logger (left) and the CR1000 (right) 91
Figure 5-27. Air temperature measurement on Feb 15, 2018: 7-hour interval (left) and 35-minut interval (right) 92
Figure 5-28. Current drain at a measurement rate of 12 s and a data rate of 5 SPS (left) and 2000 SPS (right) 93
Figure 5-29. Current drain (left) and battery life (right) at the minimal measurement rate 94
Figure 5-30. Current drain (left) and battery life (right) at the measurement rate of 1 hour 94

List of Tables

Page

Table 1-1. The abilities of commercially available data loggers 4
Table 2-1. Positive and negative voltages for VE configurations 25
Table 2-2. The voltage regulator dropout voltage 31
Table 2-3. Timing of the Logger's repetitive operations 35
Table 2-4. Current drain of the Logger's components 36
Table 2-5. The ADS1247 current drain at supply voltage of 3.3 V 37
Table 2-6. The MSP430 current drain 37
Table 3-1. The USB functions 50
Table 3-2. The ADC driver's functions 53
Table 3-3. The microSD-card functions 54
Table 4-1. The user interface command list 56
Table 5-1. Resistors for the accuracy measurements 63
Table 5-2. Statistics for measured resistances for Board 2, DR = 10 SPS, 5000 samples 68
Table 5-3. Comparison of measurements for 5000 and 510 samples 68
Table 5-4. Statistics for measured resistances for Board 1, DR = 10 SPS, 510 samples 69
Table 5-5. Statistics for measured resistances for Board 1, DR = 5 SPS, 510 samples 70
Table 5-6. Specifications of the resistors for calibration verification 72
Table 5-7. Calibration verification results 72
Table 5-8. The equivalent uncalibrated and calibrated temperature accuracies 77
Table 5-9. Specifications of the CR1000 pertaining to the half-bridge scheme with single-ended input voltages 78
Table 5-10. The CR1000 resistance measurement results 79
Table 5-11. Resistances measured with the Logger and the CR1000 83
Table 5-12. The equivalent temperature accuracy of the CR1000 and the Logger 84
Table 5-13. Resistors' values for the experiments at low temperature 85
Table 5-14. Results of the measurements at low ambient temperatures 87

Acknowledgements

I would like to thank Dr. Dejan Raskovic for his valuable advice and guiding, Dr. Vladimir Romanovsky for giving me the idea for the device and providing financial support, William Cable and Dr. Denise Thorsen for their advice and for pointing me in the right direction.

I want to thank Dr. Alexander Kholodov for providing the cooler and assisting with the organization of the cold room testing, and Mette Kaufman for setting up the cold room freezer for me.

I wish to thank the UAF writing center staff and Dr. Louise Farquharson for reviewing the monograph.

I am very thankful to Dr. Charles Mayer for his support throughout my study at the UAF. I also want to thank Carol Holtz, Reija Shnoro and Joanna Cruzan from the Office of International Programs and Initiatives for the support in organizational questions pertaining to my study in the US.

Finally, I am indebted to my family for their patience and support.

Chapter 1 Introduction

1.1 Background

With the increased focus on the global climate change, precise temperature measurements have become a crucial part of geophysical research. One of the University of Alaska Fairbanks Geophysical Institute departments, the Permafrost Lab (GIPL hereafter), studies frozen soil and the changes it experiences due to the warming climate. The study includes measuring temperatures at different depths below the surface. These measurements can be divided into four groups: near-surface, shallow, intermediate and deep measurements. Near-surface measurements are made with a 1.5 meters long thermistor probe installed in the ground. Shallow, intermediate and deep measurements are made with thermistor strings or proprietary temperature sensors that are immersed into bore holes of different depths: up to 6 meters for shallow, up to 20 meters for intermediate and up to 90 meters (60 meters in average) for deep measurements. All near-surface and shallow observatories, most of the intermediate and one of the deep observatories are equipped with dataloggers that record temperature hourly. Measurements in the deep observatories are made annually with the datalogger being lowered into the bore hole to various depth levels (typical spacing is 1 meter) [1]. Such infrequent measurements leave a gap in data on annual and interannual permafrost temperature variations [2], and ideally should be replaced with year-round measurements.

For near-surface measurements, GIPL uses the CR10X (discontinued by the manufacturer) [3] and the CR1000 [4], both made by Campbell Scientific, and Onset Computer Corporation's Hobo U12-series [5], [6] and UX120-006 loggers [7]. For shallow and intermediate measurements, mostly Hobo U12-series and UX120-006 loggers are used with a few sites equipped with the CR10X. The only automatic deep observatory is equipped with the CR10X. For annual deep measurements, the Hobo U12 Stainless Temperature Logger is used [8].

The goal of near-surface measurements is to record temperatures at the ground surface, within the active layer (the layer of ground that is subject to annual thawing and freezing [9]), at the boundary of active layer and the permafrost table, and within the upper layer of permafrost. With shallow, intermediate and deep measurements, scientists obtain the permafrost temperature profiles, determine the depth of zero annual amplitude and the depth of permafrost [1].

The depth of zero annual amplitude is defined as "the distance from the ground surface downward to the level beneath which there is practically no annual fluctuation in ground temperature." [9]. The word "practically" in this definition refers to the temperature resolution of the measuring

device. The depth of zero annual amplitude and the temperature measured at this depth depend on the geographical location, climatic and terrain conditions [9]. According to [1] and [10], for monitoring sites throughout the Arctic and Subarctic regions in North America and Eurasia, the depths of zero annual amplitude vary from about 1 to 25 meters below the ground surface, and the temperatures at these depths range from slightly below 0° C to -15° C. The temperature at the depth of zero annual amplitude serves as the indicator of interannual to decadal permafrost temperature dynamics [1]. Observations from 1977 to the present day revealed that these temperatures increase at the rate ranging from 0.01° C to more than 0.5° C per decade [10].

To determine temperature changes of 0.01° C to 0.05° C per decade within the adequate time frame, the resolution of the datalogger should be high. It should also have high accuracy in the range from -15° C to 0° C. None of the above-mentioned loggers satisfy these requirements. For the CR10X and the CR1000 the temperature resolution is improved by averaging 20 measurements taken every 3 minutes into a single hourly measurement. The drawback of this method is higher power consumption. As a result, the CR10X and the CR1000 cannot operate without an external solar battery charger. All Hobo loggers have low temperature resolution and poor accuracy, except for the Hobo UX120-006, which has high resolution but poor accuracy. Hobo U12-series loggers do not have the ability to do intermediate measurements, neither do they have enough memory to keep more measurements if one wants to do post-averaging after the data collection. Averaging time during the annual deep measurements cannot be too long because of the overall duration of the measurement procedure. Scientists allow the logger to equilibrate with the environment for 2-3 minutes, then wait for another minute for the logger to measure the temperature and move to the next depth. For a bore hole of 60 meters depth it takes 4 hours to complete the measurement.

All Hobo loggers in use by GIPL, except for the U12-008 outdoor/industrial logger, have operational issues when working at temperatures below -20° C. Additionally, the 2.5 mm stereo-type connector does not secure the cable tightly, which leads to sensor fails. Furthermore, Hobo loggers have a small number of channels, so scientists need to install several of them at one site.

GIPL operates around 150 monitoring sites in Alaska, Canada and Russia. However, due to an uneven spatial distribution of the sites, large arctic and subarctic areas in Canada, Greenland and Russia are not covered, leaving gaps in knowledge on the changes in permafrost happening in these regions. In the future, GIPL scientists are planning to increase the number of monitoring sites [10]. None of the loggers in use by GIPL makes an ideal candidate for new installations. Campbell Scientific loggers are expensive, large and heavy. At minimum, they require an automotive type battery, a special waterproof

box and a mast to mount solar charger panel. This adds significantly to the installation and transportation cost. Another factor to consider is the frequency of visiting the sites which will have to be decreased from the current once per year to at least once per two years. In this case, the typical battery life, and, for some models, the storage capacity of Hobo loggers is not sufficient. I conducted the research on the commercially available loggers, which showed that there are no loggers on the market that will meet all the requirements posed by GIPL scientists.

All the above-mentioned factors drive the need to develop a new data logger, which can resolve small changes in temperature and is accurate, inexpensive, small, and has enough memory capacity and battery life to sustain through the long logging sessions. At the same time, the logger should be suitable for all four types of soil temperature measurements. The prototype of such data logger has been developed and is presented in this thesis.

1.2 Overview of Commercially Available Loggers

Table 1-1 (page 4) summarizes the information on GIPL requirements and the abilities of the most appropriate commercial loggers. The following tendency can be inferred for the commercially available devices: loggers that meet the requirements in terms of temperature resolution/accuracy, channel number and memory capacity, are usually large, heavy, not waterproof, and have high power consumption. Conversely, loggers that are small and meet the battery life requirement usually have poor temperature resolution or accuracy and a small number of channels. The data logger, presented in this thesis, was designed and manufactured with the intention to combine the above-mentioned features in one device.

1.3 Objectives

The objective of the proposed thesis is to design, implement, and test the prototype of a multi-channel outdoor data logger with wireless capability (the Logger). The Logger should meet requirements posed by GIPL, however it should also be adaptable so that it can be used by any researcher. The Logger's performance should be verified by an extensive testing, focusing on the applications of interest to GIPL researchers.

The designed Logger should have the following features:

1. Compatibility with the existing thermistor probes and strings that use 10 kΩ nominal resistance thermistors.
2. A multi-channel ADC (not fewer than 16 channels).
3. The temperature data resolution of at least 0.01° C.

4. The accuracy of at least 0.01° C near 0° C and less than 0.05° C elsewhere in the operating range of -40° to 40° C when making measurements with thermistors.
5. Maximum temperature range of -40° to +40° C.
6. The ability to store hourly measurements from all channels for at least 2 years.
7. Battery powered with battery life of at least 2 years.
8. The Logger should allow user to update firmware in the field without special equipment.
9. The ability to download data wirelessly and potentially participate in a wireless sensor network.
10. The Logger size is less than about 3 cm in diameter.
11. It should be less expensive than the CR1000, which costs around $1500.

Table 1-1. The abilities of commercially available data loggers

Parameter	Require-ment	Does a logger meet a requirement Yes/No?						
		Onset Computer Corp. HOBO U12-008 [6]	Onset Computer Corp. Hobo UX120-006 [7]	Campbell Scientific, Inc. CR1000 [4]	Campbell Scientific, Inc. CR1000X [11]	Accsense, Inc. VersaLog DCV-2 [12]	Grant Instru-ments SQ2040 [13]	Thermo Fisher Scientific DataTaker DT85W [14]
Number of channels	16	No	No	Yes	Yes	No	Yes	Yes
Temperature resolution, °C	0.01° C	No	Yes	No	Yes	No	Yes	Yes
Accuracy (near 0 °C), °C	0.01° C	No	No	No	Yes	No	Yes	Yes
Operational temperature range, °C	-40 to 40	Yes	No	Yes	Yes	Yes	No	Yes
Memory Size	2 years of measure-ments	No	Yes	Yes	Yes	Yes	Yes	Yes
Battery Life	2 years	No	Yes	No	No	Yes	No	No
Water-proofness	Waterproof	Yes	No	No	No	No	No	No
Form factor	Small size	Yes	Yes	No	No	Yes	No	No
Wireless connectivity	Required	No	No	No	No	No	Yes	Yes

1.4 Overview of the Designed Logger

For temperature measurements, the Logger uses a 24-bit sigma-delta ADC with thermistors as sensing elements. The ADC ensures high resolution and accuracy of temperature measurements. An analog signal multiplexer is used to connect 16 thermistors to the ADC. The heart of the system is the Texas Instruments, Inc. MSP430F5659 microcontroller [15]. It controls the ADC and other peripherals, manages data, provides an interface to the PC, etc. The Logger uses two types of memory: the microcontroller's flash memory and a microSD card of up to 2 GB. The 512 KB flash memory can store 7680 measurements for each channel, while a 2 GB SD-card provides an additional storage space for 41.9 million measurements for each channel. Several power conservation features are implemented in the Logger. They include but are not limited to: the microcontroller and peripherals are placed into a low power mode when inactive; the SD-card supply voltage is cut off with a power switch; the Logger is powered from the PC when connected over USB; all computationally-intensive data processing is performed on the PC. The Logger's user interface is implemented in MATLAB. It is a command-line interface, which provides full control over the Logger.

1.5 Overview of the Results

Extensive tests were conducted with two Loggers and the CR1000 to determine and compare the performance of all three devices. The CR1000 was chosen as a competition to the Logger because it is the most widely used logger in academia. The CR1000 is a versatile device capable of making different kinds of measurements. Therefore, when comparing the CR1000 to the Logger, only its temperature measuring ability is considered.

The Logger's design requirements are: resolution of 0.01° C; the accuracy of 0.01°C near 0° C and less than 0.05° C elsewhere in the operating range -40° to 40° C when making measurements with thermistors. Ideally, the Logger with connected thermistors should have been calibrated against a thermometer that has been previously calibrated as a secondary standard. In this case, the Logger's accuracy would have been referred to the international temperature scale of 1990 (ITS-90) [16]. However, since no such thermometer was available, the Logger was calibrated using the data obtained from measuring a known resistive input. By converting measured resistances to temperatures, a so-called equivalent temperature accuracy was determined for the Logger. Consequently, this accuracy is referred to the ohmmeter and not the ITS-90. The determined uncalibrated equivalent accuracy is below 10 mK within -25° C to 40° C and below 20 mK elsewhere. The developed calibration technique provides the equivalent accuracy below 0.3 mK within -40° C to 40° C. Measured resolution is below 2.5 mK in the

entire temperature range. In the experiments, both Loggers showed similar results with the difference being attributed to the components' tolerances. For the CR1000, the measured equivalent temperature accuracy is 0.1° K at 0° C with the worst-case value of 0.25° K.

To determine the effect of ambient temperature on the accuracy of resistance measurements, the series of experiments was conducted with the Logger operating at -30°, -25°, -20° and 6° C. It was noted that with the ambient temperature being lowered, the accuracy of resistance measurements was increasing.

Finally, the Loggers and the CR1000 with connected thermistors were tested outdoors to see if they would follow ambient temperature changes in the same way. The ambient temperature varied within -28° C to 0° C during the experiments. All devices showed similar results. Measured values followed the air temperature measurements obtained from local weather stations. As expected, the Logger demonstrated superior temperature resolution compared to the CR1000.

Measurements of the Logger's power consumption showed that the expected battery life exceeds 8 years for a measurement rate of 1 hour and any data rate.

1.6 Competitive Advantages of the Logger

In terms of temperature measurements, the Logger performs better than the CR1000 and is expected to perform about the same as the CR1000X. The CR1000 was one of the best commercially available loggers on the market at the time the Logger's design started, and the CR1000X is a superior model to the CR1000, which was released in August 2017. The advantages of the Logger over the CR1000/CR1000X are smaller physical dimensions and weight, better power efficiency, and smaller price. Because of smaller size and weight, the Logger has considerably lower transportation and installation costs than the CR1000/CR1000X. For installation, the CR1000/CR1000X requires, at minimum, a waterproof box, an automobile type battery, a solar panel for recharging the battery and a mast on which to mount the panel.

Most commercially available loggers have one to four channels, but this Logger has 16. Most commercially available loggers with 16 channels have higher power consumption, lower memory capacity, and larger physical dimensions and weight than the Logger. Some of them either cannot work outdoors or have problems operating at low ambient temperature. Despite having 16 channels, this Logger is small, has large memory capacity and long estimated battery life. With an appropriate casing or if covered with epoxy it can work outside down to -40° C.

The hardware and firmware architectures allow great flexibility. Thermistors with different nominal resistance (20 kΩ, 10 kΩ, and 5 kΩ) can be used as sensing elements. Conversion rates from 5 to 2000 samples per second (SPS) and different gain settings for the output signal are possible. The Logger can operate based on the real time clock schedule or non-stop. The firmware can be upgraded in the field.

1.7 Comparison to Similar Devices Developed in Academia

In this section the Logger is compared to two precision thermometers developed in academia in the past, namely a fast multichannel precision thermometer [17] (Thermometer 1), and a fast micro-Kelvin resolution thermometer based on NTC thermistors [18] (Thermometer 2). The two thermometers were built with high temperature accuracy and resolution in mind; they both use thermistors as sensing elements, and they are multichannel. Therefore, comparing the Logger with these thermometers is appropriate.

The common feature of these devices is that they are thermometers, and not data loggers. Instead of storing data into internal memory, they stream it to a host PC over a communication interface: UART for RS232 interface for Thermometer 1 or the USB interface for Thermometer 2. Additionally, the thermometers are not battery powered. Thermometer 1 is powered from the dedicated power supply, while Thermometer 2 is powered from the host PC over the USB. Because of this, the questions of power consumption and power conservation are not discussed in [17] and [18].

Thermometer 1 has 32 analog channels. Unlike the Logger, which utilizes an analog multiplexer to switch between its 16 channels, it has 32 analog front-ends and four ADCs. The ADC's work sequentially; they are controlled over an SPI bus by a single microcontroller. Thermometer 2 has four channels, which are the part of the onboard ADC unit. Both thermometers use a simple voltage divider circuit to sense temperature with 30 kΩ thermistors, which are a BetaTHERM 30K6A1A type [19]. Thermometer 1 has an operational temperature range of -50° C to 40° C, while Thermometer 2 operates from -30° C to 30° C. This is close to the Logger's range of -40° C to 40° C.

The 30K6A1A thermistor has a power dissipation constant of 0.75 mW/°C. For Thermometer 1, the stated accuracy of the device is 25 mK in the range from -50°C to 10°C. To achieve this accuracy, the dissipated power should be smaller than 18.75 µW. The design limits the thermistor dissipated power at 15 µW, which does not violate the 18.75 µW limit.

For Thermometer 2, the declared accuracy is 2.5 mK within the range of -30° C to 30° C. Authors also state that the thermistor dissipated power is 40 µW at 25° C. However, to achieve 2.5 mK accuracy,

the dissipated power should be smaller than 1.875 μW. With the dissipated power of 40 μW, the thermistor's self-heating will cause the accuracy to decrease by 53.3 mK. Thermometer 2 is used to measure temperature of liquids, which in general conduct heat better than the air. Presumably, the authors took this fact into account when designing the sensing circuit, however they never stated it explicitly. The Logger, on the other hand, was developed to support different applications, with air temperature measurements being one of those applications. Therefore, the thermistor dissipated power within the entire temperature range is smaller than 10 μW, which corresponds to a temperature accuracy of 0.01° C or higher.

Thermometer 1 with connected thermistors was calibrated in a thermostat that can provide a long-term stability of 3 mK. The calibration points from -60° C to 40° C with a 10° C step were taken during the calibration. The same procedure but with 5° C steps was used to obtain the Thermometer 1 accuracy. Additionally, Thermometer 1 noise voltage was determined by simulating the thermistor with resistors representing the temperature range from -37° C to 35° C. Thermometer 2 with connected thermistors was calibrated against the calibrated thermometer. Thermometer 2 noise voltage was also determined by simulating the thermistor with four resistors.

The Logger's noise voltage was determined by replacing the thermistor with 17 resistors representing the temperatures in the range from -40° C to 40° C and measuring the resistors with the Logger. By measuring the same resistors with an ohmmeter, converting resistance data from both measurements into temperatures, and calculating the difference between these temperatures, the equivalent temperature accuracy of the Logger was determined, which was smaller than 20 mK.

The resistances measured with the Logger and the ohmmeter were used to develop a calibration technique, which was utilized when measuring four additional resistors. In this case, the equivalent accuracy was smaller than 0.3 mK.

There are few more differences between the Logger and the Thermometers. First, the maximum measurement rate for Thermometer 1 is 15 SPS, for Thermometer 2 is 10 SPS, while for the Logger it can vary from 5 SPS to 2000 SPS. Second, the Logger and Thermometer 1 are using a real-time clock for supporting time-stamped measurements. For Thermometer 2, authors do not mention such ability.

1.8 Thesis Organization

Chapter 2 describes hardware architecture including the comparison of sensing schemes, the choice of parts, the design of a power supply circuitry, and the design of a schematic and a printed circuit board (PCB).

Chapter 3 describes the Logger's firmware, including the details of source code functions and peripheral drivers.

Chapter 4 describes the user interface and its functionality.

Chapter 5 provides details on the experiments that were conducted to determine the Loggers' performance and compare it to the CR1000 and presents results of those experiments.

Chapter 6 gives a conclusion and a summary of current accomplishments and explains the future work required to improve the Logger.

Chapter 2 Hardware Architecture

The Logger's hardware block-scheme is presented in Figure 2-1. The block-scheme is used to guide the discussion of the hardware architecture throughout this Chapter.

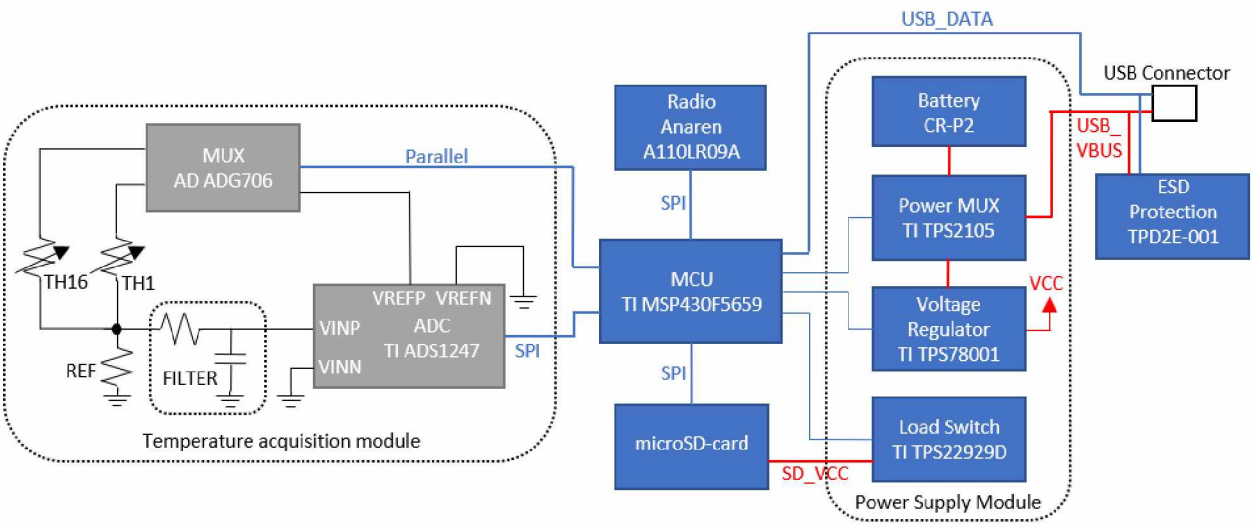

Figure 2-1. The Logger's hardware block-scheme

The temperature acquisition module measures voltage across a sensing element, converts the measured voltage into a binary-coded value, and sends the conversion result to the microcontroller. Section 2.1 of this Chapter discusses the selection of the sensing circuit, an analog multiplexer, and an analog-to-digital converter (ADC) for the temperature acquisition module. It also provides the estimation of the module's temperature resolution.

The power supply module supplies regulated voltage to all components of the Logger, selects the primary supply source, and shuts down the microSD-card when not in use. The module consists of a battery, a power multiplexer, a voltage regulator, and a load switch. The selection of the components and the estimation of current drain and battery life are provided in section 2.2 of this Chapter.

Other supporting hardware includes a microcontroller unit (MCU), a microSD-card, a radio, and an electrostatic discharge (ESD) protection integrated circuit (IC). These components are set up as suggested in their respective datasheets and are not covered in many details.

The manufactured Logger, the schematic and the printed circuit board (PCB) of the Logger are presented in section 2.3, Appendix A, and Appendix B respectively.

2.1 Design of a Temperature Acquisition Module

Because the temperature measurements are the highest priority for the Logger, the design started with a temperature acquisition module (Figure 2-1). The essential parts of this module are a sensing circuit, a signal conditioning circuit and an ADC. The sensing circuit includes sensing elements, a reference resistor, and an analog multiplexer. The analog multiplexer allows connecting 16 channels to the single channel of the ADC. The signal conditioning circuit consists of an anti-aliasing filter and an amplifier. The anti-aliasing filter is a low-pass filter that attenuates high-frequency noise, which can be translated to low-frequency noise during the sampling process and, as a result, can appear in the ADC's output data. The amplifier removes the loading effect of the ADC's input resistance and can optionally amplify the input signal.

2.1.1 Schemes for a Sensing Circuit

Several different schemes can be utilized for the sensing circuit. Among them are two-, three-, four-wire current and voltage excitation schemes [20], [21], as well as two-, three-, four- and six-wire bridge schemes [22]. However, for the Logger to be compatible with the existing probes (the schematic diagram is presented in Figure 2-2), the choice of the sensing circuit has been reduced to two-wire schemes with a thermistor as the sensing element. Therefore, from this point on, only two-wire schemes are discussed.

The existing GIPL probes use U.S. Sensor Corp. PS103J2 thermistors [23]. They have a nominal resistance of 10 kΩ at 25° C and accuracy of ±0.1° C. The manufacturer claims that any two thermistors have a temperature discrepancy not more than 0.1° C, which makes them 0.1° C interchangeable.

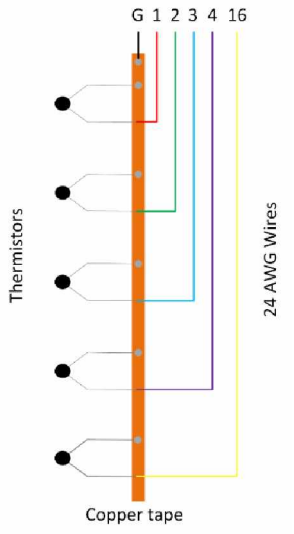

Figure 2-2. A thermistor probe schematic diagram

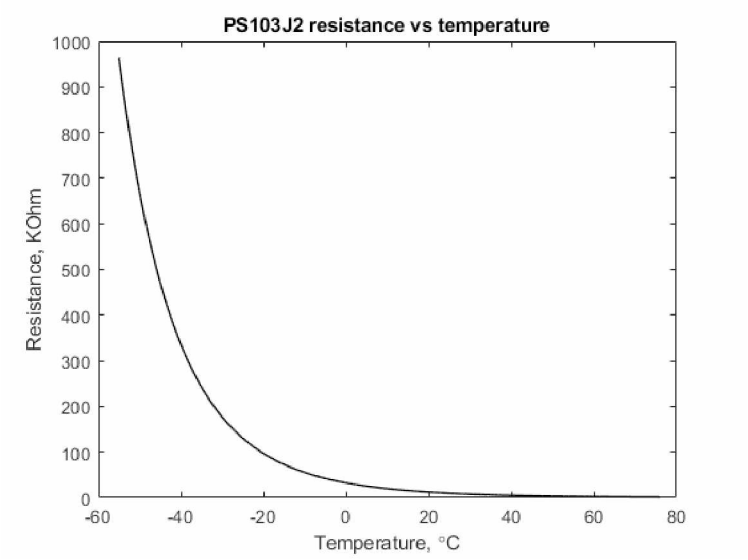

Figure 2-3. The PS103J2 10 kΩ thermistor resistance versus temperature

A resistance versus temperature (RT) data for the PS103J2 thermistor is provided by the manufacturer in a tabulated form. The range of temperatures spans from -55° C to 76°C, which corresponds to the range of resistances from 964 kΩ to 1.4 kΩ, the temperature resolution of data is 0.05° C. The RT-curve generated from the PS103J2 thermistor RT-data is shown in Figure 2-3. Since the range of interest for the Logger is - 40° C to 40° C, most of the figures in thesis present data in this range and not in the range of the thermistor's RT-data. For - 40° C to 40° C, the PS103J2 thermistor resistance changes from 336.5 kΩ (-40° C) to 5.3 kΩ (40° C).

2.1.2 The Tentative Study of the Schemes for the Sensing Circuit

Because the thermistor resistance changes nonlinearly with temperature, voltage across the thermistor changes nonlinearly, too. For this reason, a voltage difference between adjacent temperature readings (voltage resolution) is not constant and depends on the slope of thermistor's RT characteristic. The minimum value of the voltage resolution defines the maximum usable value of the ADC's least significant bit (LSB) and, consequently, a minimum bit resolution of the ADC. A tentative study of a voltage excitation (VE) scheme, a current excitation (CE) scheme and a bridge scheme was performed to determine the approximate value of the minimum voltage resolution. I used OrCAD Capture/PSPICE (PSPICE hereafter) and MATLAB software packages to examine the circuits' operation. In PSPICE, the primary sweep functionality was used to simulate the PS103J2 thermistor; the resistance values were varied from 700 kΩ to 1 kΩ with the step of 1 kΩ. In Matlab, the PS103J2 thermistor RT-data was used.

First, the VE scheme was analyzed. This scheme can have four configurations. They are presented in Figure 2-4 and are labeled with numbers 1 through 4. The VE scheme (any configuration) is a simple voltage divider circuit consisting of a reference voltage source and three resistors, which represent multiplexer ON (Rmux), thermistor (Rtherm), and reference (Rref) resistances. Resistance of wires (Rwire) is negligible and, therefore, is omitted from the discussion. Instead of replicating the sensing and signal conditioning circuits for each channel, I decided to use an analog multiplexer. The multiplexer is characterized with ON resistance – the resistance of the closed switch path between the drain and source terminals [24]. Constant values were used for the reference voltage and multiplexer ON resistance; 2 V and 4 Ω were chosen as common values for these parameters. The ADC's input resistance was not accounted for because at the time I considered it was infinite.

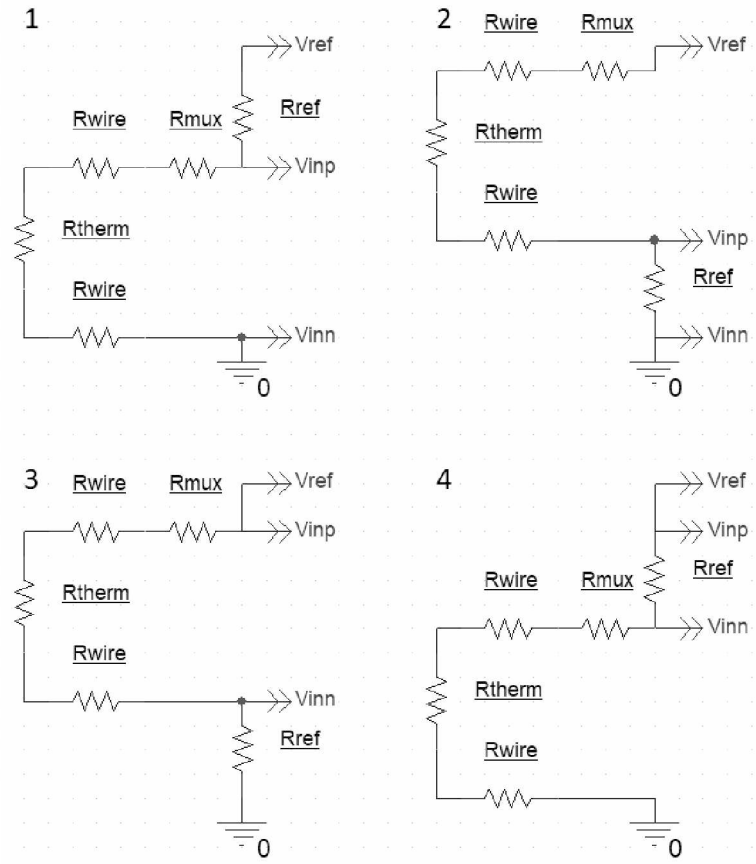

Figure 2-4. Four configurations of the voltage excitation (VE) scheme

Configurations #1 and #2 provide a single-ended input signal to the ADC, while #3 and #4 provide a pseudo-differential signal. Single-ended signal uses 0 V as a reference. For a pseudo-differential signal [25], only a negative voltage changes, while a positive voltage is held constant at

reference voltage level. In Figure 2-4, “Vinp” and “Vinn” denote positive and negative input voltages to the ADC.

Since the small thermal mass of the thermistor makes it very susceptible to self-heating [22], it was necessary to make sure that this effect would not compromise the accuracy of the Logger. According to the manufacturer, PS103J2 thermistor has a dissipation constant of 1 mW/°C. Therefore, to achieve 0.01°C accuracy, thermistor power should be limited to 10 µW throughout the entire range of thermistor resistances. Utilizing the PSPICE secondary sweep feature, it was determined that the reference resistance value of 100 kΩ or larger would maintain the constraint of 10 µW (Figure 2-5). Therefore, in subsequent simulations, the reference resistance value of 120 kΩ was used.

In this thesis, for all figures, where thermistor resistance is used as the X-axis, the X-axis is reversed to correspond to temperature axis spanning from negative to positive values.

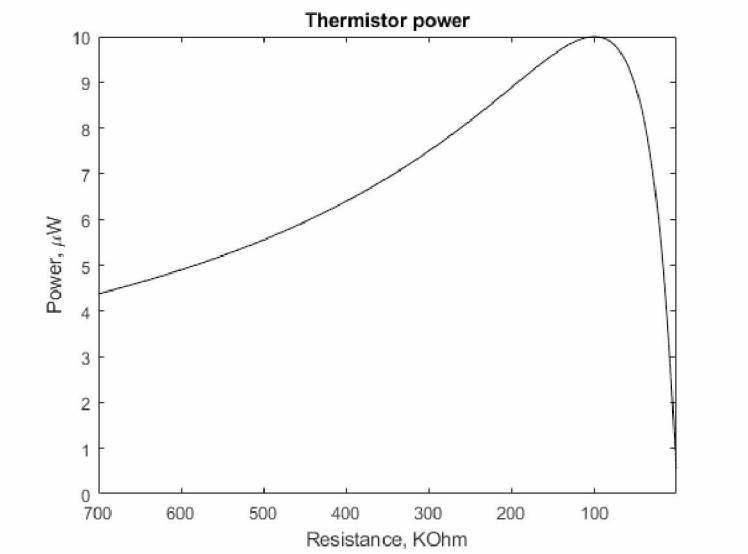

Figure 2-5. The thermistor’s dissipated power with Rref = 100 kΩ

As discussed in the beginning of this section, the minimum value of the ADC’s input voltage resolution determines the maximum usable value of the ADC’s LSB. The input voltage resolution can be calculated as the difference between the adjacent voltage values. The PS103J2 RT-data was used to calculate the input voltage for all VE configurations; as an example, the input voltage for the VE configuration #1 with the reference resistor of 120 kΩ is presented in Figure 2-6.(left). However, the thermistor’s RT-data has a temperature resolution of 0.05° C, while the required resolution for the Logger is 0.01°C. Therefore, to determine the input voltage resolution that will maintain the 0.01° C temperature resolution, it was necessary to increase the number of data points by five. To achieve this,

MATLAB curve fitting tool was utilized. The input voltage data was fitted with the 8th order Fourier equation; the root mean squared error was 1.6 µV, making it a very good fit. The Fourier equation was used to generate the input voltage data with the larger number of points, from which the voltage resolution required for the 0.01° C temperature resolution was obtained (Figure 2-6.(right)). The magnitude of the minimum voltage resolution was equal to 35 µV for all VE configurations. Alternatively, the same task could have been achieved by upsampling the voltage data and using interpolation to obtain the intermediate values.

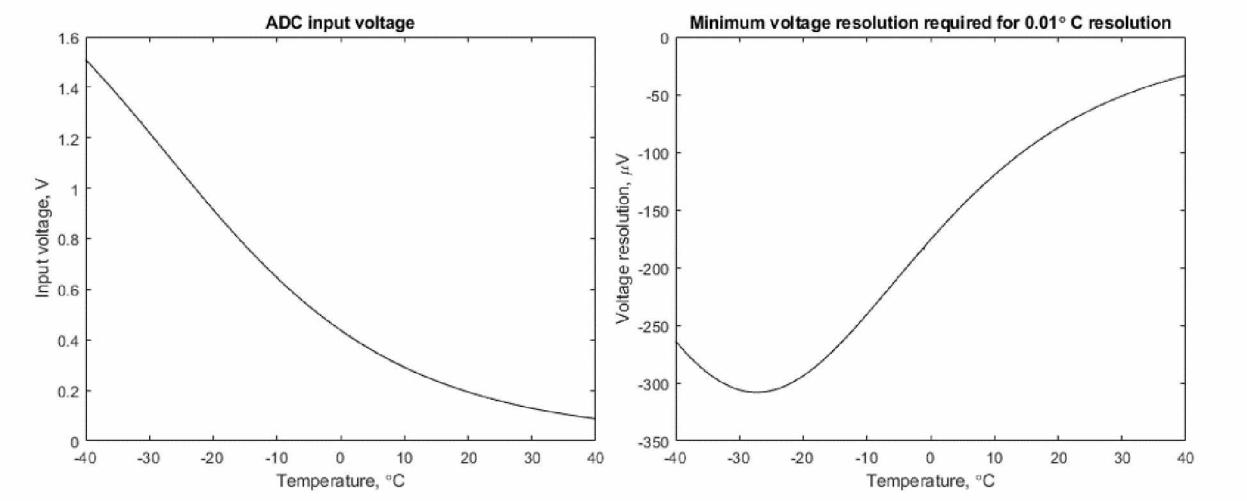

Figure 2-6. VE configuration #1: ADC input voltage (left) and voltage resolution required for 0.01° C temperature resolution (right)

Next, the bridge scheme was studied (Figure 2-7). A single varying element bridge is the most suitable bridge scheme for the application utilizing thermistors [22]. In this configuration one of the bridge branches contains a thermistor (Rtherm), while the other three branches contain fixed resistors (R) of the same value. The ADC measures a voltage difference between the center points of two voltage dividers connected across the excitation voltage. To limit the thermistor's dissipated power, 120 kΩ fixed resistors were used.

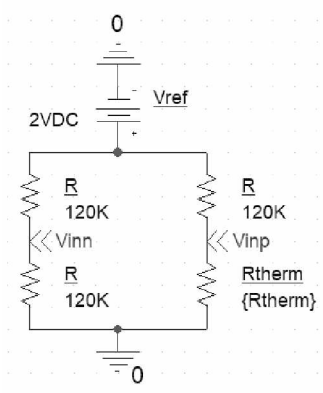

Figure 2-7. A simple bridge scheme

The calculated minimum voltage resolution (Figure 2-8.(right)) was 35 µV – the same as for the VE scheme, which was expected because both schemes had the same value resistors.

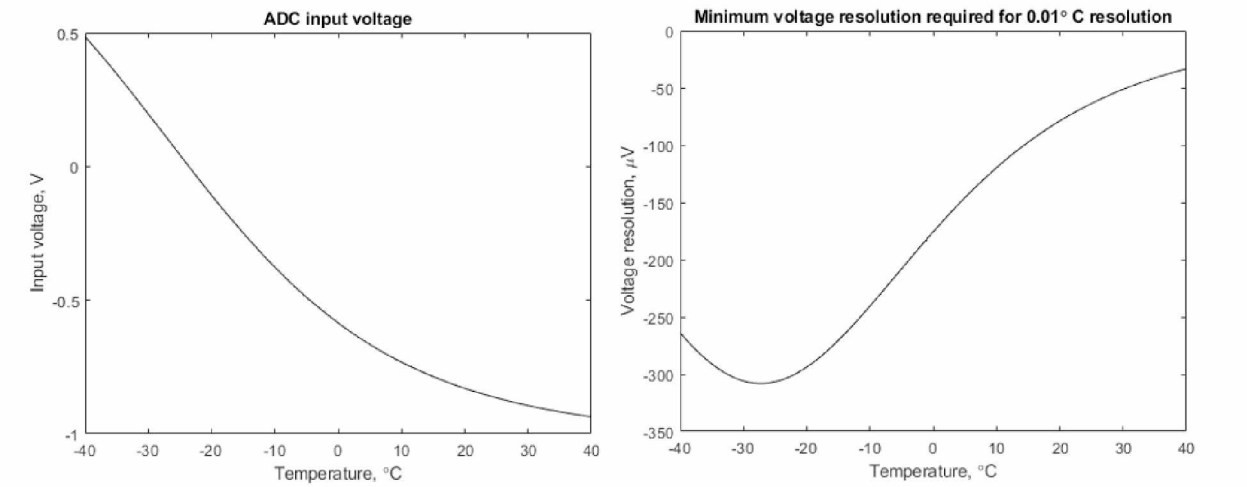

Figure 2-8. Bridge scheme: ADC input voltage (left) and voltage resolution required for 0.01° C temperature resolution (right)

A differential input voltage to the ADC becomes negative for any thermistor resistance values smaller than 120 kΩ (Figure 2-8.(left)). Therefore, implementing the bridge scheme in this configuration will require a bipolar supply voltage for the ADC. To avoid negative voltage values, the fixed resistors value should be smaller than the minimum thermistor resistance. In this case, however, two resistors, as discussed in [26], should be added to the scheme to limit the thermistor's dissipated power. The resistors' values should be properly chosen to set the common mode of the signal conditioning amplifier. Adding these two resistors to the circuit makes the ADC's input voltage and the voltage

resolution smaller than in the case of the simple bridge scheme. Consequently, the input voltage requires amplification.

Finally, the CE scheme was evaluated. The analysis was built around the circuit that includes a current source (Iset), a thermistor (Rtherm), and a reference resistor (Rref) (Figure 2-9). In Figure 2-9, labels “Vrefp” and “Vrefn” denote positive and negative voltages supplied to the ADC’s reference input. In this circuit, the source’s set current value determines the thermistor’s dissipated power, while the combination of the set current and the reference resistor values determines the ADC’s input and reference voltage values. The input voltage resolution is directly proportional to the input voltage range.

The analysis was restricted by two factors. First, the thermistor’s dissipated power had to be smaller than 10 µW. The dissipated power has the highest value at the thermistor’s highest resistance. The resistance of the PS103J2 thermistor at -40° C is 336.5 kΩ. Therefore, for this resistance value, the set current had to be smaller than 5.5 µA. Second, the compliance voltage of the current source had to be considered. Compliance voltage determines the maximum voltage available to the connected load. Typically, it varies from 0.7 to 0.9 V. Assuming the compliance voltage of 0.7 V, voltage across the series connection of the thermistor and the reference resistor had to be smaller than the supply voltage of 3.3 V minus 0.7 V, which is equal to 2.6 V.

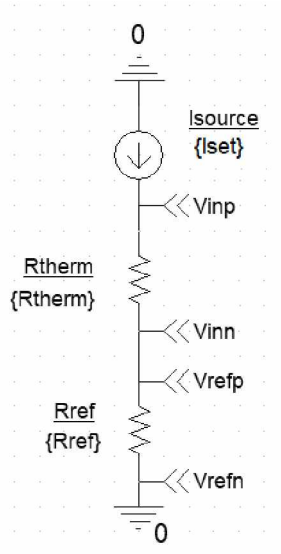

Figure 2-9. A simple current excitation (CE) scheme

The analysis objective was to determine the set current and the reference resistor values that would allow meeting the above-mentioned requirements and provide reasonable values for the ADC’s input and reference voltages. The word “reasonable” in this case means “as large as practically possible”. Studying different combinations of set currents and reference resistors, I found that the set

current of 3 µA and the reference resistor of 475 kΩ would satisfy the analysis objectives. In Figure 2-10, the red solid line shows the maximum voltage available for the series connection of the thermistor and the reference resistor due to the compliance voltage requirement, and the black dashed line shows the voltage across the thermistor and the reference resistor. The two lines intersect at -42.25 °C thus setting the limit on the operating temperature of the scheme (showed in figure with the red dotted line). The reference voltage (the black dotted line) is 1.425 V. The input voltage (the black solid line) is 1.010 V at -40° C and 0.016 V at 40 °C.

In the range from -28° C to 40° C, the CE scheme with the reference resistor of 475 kΩ has smaller voltage resolution (the difference between the adjacent voltage values), than both the VE and the bridge schemes (Figure 2-11). For example, at 0° C the voltage resolution is 50 µV for the CE scheme and 176 µV for the VE and the bridge schemes. The minimum voltage resolution is 6.4 µV at 40° C for the CE scheme and 35 µV at 40° C for the VE and the bridge schemes. However, the CE scheme's small voltage resolution can be improved by amplifying the ADC's input voltage.

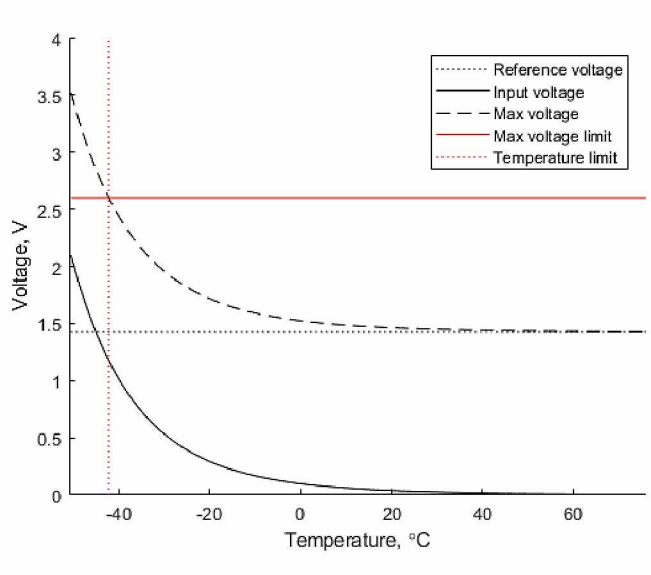

Figure 2-10. CE scheme: the ADC voltages

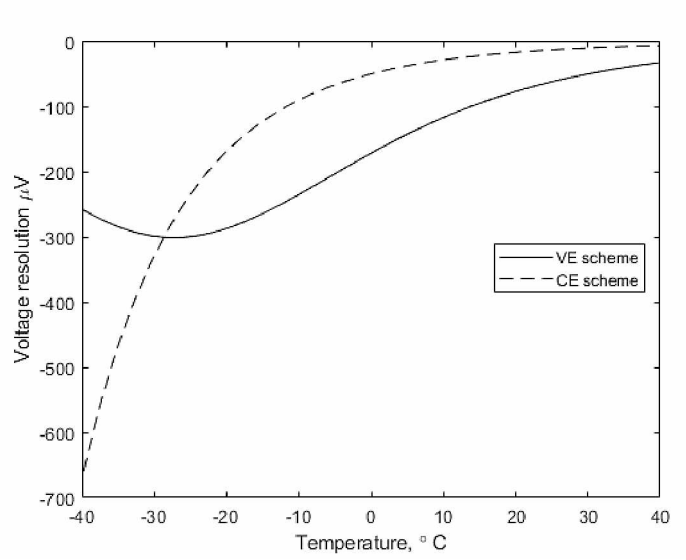

Figure 2-11. Voltage resolution of the CE and the VE schemes

The amplification can be achieved by using a programmable gain amplifier with an appropriate gain setting. Tentatively, the input voltage can be multiplied by two for thermistor resistances below 230 kΩ (temperature range above -34° C), by four for resistances below 110 kΩ (temperature range above -23° C), by eight for resistances below 59 kΩ (temperature range above -11° C), and by 16 for resistances below 29 kΩ (temperature range above 2° C). To determine the exact resistance ranges where specific multiplication ratios can be applied, it is necessary to determine the amplifier's common-mode voltage and compare it to the common-mode voltage requirements at the respective gain settings. For these reasons, the CE scheme's minimum voltage resolution of 6.4 μV (no amplification) was not used to determine the ADC's maximum LSB value.

The preliminary examination of the three schemes allowed me to determine the minimum voltage resolution required to achieve the 0.01° C temperature resolution and make conclusions on the applicability of different schemes for the Logger. Next step in the design process was the ADC selection.

2.1.3 The ADC Selection

Without defining the selection criteria, the task of choosing a proper ADC would have been daunting. To limit the number of possible ADC options, the following parameters were used:

1. Resolution in bits.
2. Single-ended power supply voltage of less than 5 V.
3. Power consumption less than 5 mW.
4. Digital interface either SPI, I2C or UART.

5. Number of inputs from 1 to 4.
6. Operating temperature range from -40° C to 40° C or wider.
7. Small outline integrated circuit package (HTSSOP, SSOP, TSSOP, VSSOP, SOT-23, and SOIC) – this would have allowed hand-soldering in case it was needed.

The formula for an LSB calculation can be used to determine the ADC's minimum bit resolution, as shown by Eq.2.1:

$$LSB = \frac{V_{REF}}{2^N - 1} \quad 2.1$$

In Eq.2.1, V_{REF} is the reference voltage and N is the number of resolution bits of the ADC.

Calculations were performed for V_{REF} value of 2 V. Comparing the resultant LSB values with the minimum voltage resolution of 35 µV, the minimum bit resolution was determined to be 16 (LSB equals 30.5 µV). Therefore, the first criteria became 16 or more bits of resolution.

To choose the best ADC for the application, I used the parametric search engines at the Texas Instruments, Inc. and Analog Devices, Inc. websites. Later, only Texas Instruments ADCs were considered for implementation in the Logger – there were plenty of appropriate ADC models available and, at a glance, Texas Instruments ADCs were better documented.

The parametric search based on the above-mentioned criteria allowed me to narrow down the number of ADC options from 1000 to 29 units. The search result included 14 24-bit sigma-delta (Σ-Δ) ADCs, one 20-bit and two 18-bit Σ-Δ ADCs for bridge measurements, 11 16-bit Σ-Δ ADCs and one successive approximation (SAR) 16-bit ADC. All 16-bit ADCs were excluded from further consideration because of the large selection of higher resolution ADCs. Additionally, the 16-bit SAR ADC did not have auxiliary signal conditioning modules, compared to Σ-Δ ADCs. Almost all 24-bit Σ-Δ ADCs had the same value of the integral linearity error (INL); therefore, this parameter was not accounted for when choosing the ADC. All 24-bit Σ-Δ ADCs were of the "no missing code" type.

Next, Σ-Δ ADCs without a programmable gain amplifier (PGA) and a voltage reference were excluded. An embedded PGA makes up an almost complete signal conditioning circuit. Moreover, input-referred noise specifications of the PGA are provided in the datasheet. This makes it possible to estimate the resolution of the future device before even implementing it. The internal voltage reference allows implementing the voltage excitation scheme without extra components.

Internal PGAs of 20-bit and 18-bit Σ-Δ ADCs for bridge measurements have only two gain settings – either 64 or 128. Unfortunately, neither setting is appropriate for any of the sensing circuit schemes being reviewed. Therefore, these ADCs were removed from further consideration.

The Logger is required to accommodate the needs of many researchers, therefore only ADCs with multiple options for the data rate were considered. Those ADCs can operate at smaller data rates with better resolution or at faster data rates with degraded resolution.

Consequently, only three ADC units were left: the ADS1246 (single input), the ADS1247 [27] (same as the ADS1246 but with four inputs) and the ADS 1220 (four inputs) [25]. Because the ADS1247 and the ADS1220 has four inputs, they were the main candidates for the implementation in the Logger. To choose between these units, I focused more on their performance characteristics.

Both units are specifically designed for measurements with thermistors and other temperature sensors. Besides a PGA and a voltage reference, they have an ability to switch between an internal and an external voltage reference, a digital filter with an excellent 60 Hz rejection, and a functionality for self and system calibration. Both the ADS1247 and the ADS1220 support single-ended and differential signals. This feature was deemed useful at the time of the ADC selection, because no decision was made regarding the sensing circuit scheme – the current excitation and the bridge schemes required a differential input signal, while the voltage excitation scheme could make use of both single- and differential-ended signals.

Although both ADCs are very similar, the ADS1247 stands out because of its superior characteristics pertaining to DC measurements. Compared to the ADS1220, the ADS1247 has much smaller input offset voltage, gain error, input-referred noise and significantly better power-supply rejection (these are, essentially, the PGA related characteristics). Additionally, it has more stable internal reference voltage. For these reasons, the ADS1247 was chosen for the Logger.

Next, the analog multiplexer was chosen for the Logger. The following criteria were used during the selection process:

1. 16 channels.
2. Low ON resistance and resistance match between channels.
3. Single-ended power supply voltage of less than 5 V.
4. Low power consumption.
5. Digital or parallel communication interface.
6. A package type suitable for hand soldering.

The only analog multiplexer that met all criteria was Analog Devices ADG706.

2.1.4 The Sensing Scheme Selection

As mentioned earlier in this Chapter, three schemes were considered for the Logger's sensing circuit, namely the voltage excitation (VE) scheme, the current excitation (CE) scheme, and the bridge scheme. Once the ADC and the analog multiplexer were chosen, I studied the schemes again, now taking into the account their respective specifications.

The VE scheme was evaluated the same way as described in 2.1.2. The voltage reference and the multiplexer ON resistance values were set to 2.048 V and 4.25 Ω respectively, as specified in their datasheets. To minimize the error associated with the reference resistor drift, I chose a Vishay Y1441116K667A0L 116.667 kΩ resistor with 0.005% initial tolerance and ± 0.2 ppm/°C temperature coefficient (TCR) within the range from -40° C to 40° C as a reference resistor. The calculated input voltage resolution was almost identical to the one obtained in 2.1.2, which was expected since the parameter values were close to the ones used in the tentative study in 2.1.2.

The VE scheme (as well as the other two schemes) can be considered a ratiometric circuit. For the ratiometric circuit, the accuracy of measurements depends only on the accuracy of the reference resistance and not the excitation voltage or current. In the case of the VE scheme, this is achieved by providing the same voltage to both the sensing circuit and the ADC's reference input. The only constraint imposed on the reference voltage source is that it should have an excellent short-term stability. The ADS1247 has a very-low drift voltage source, which meets this requirement.

Using configuration #3 (Figure 2-4.(3)) as an example, measured resistance R_{MEAS}, which includes the thermistor, the multiplexer ON, and the connecting wires' resistance can be calculated using formulas presented in Eq. 2.2 – 2.4. Voltage V_{MEAS} across the measured resistance R_{MEAS} is calculated using the voltage-divider formula:

$$V_{MEAS} = \frac{V_{REF} * R_{MEAS}}{R_{MEAS} + R_{REF}} \qquad 2.2$$

The same voltage can be calculated as

$$V_{MEAS} = code * \frac{V_{REF}}{2^N - 1} \qquad 2.3$$

where $code$ is the binary-coded interpretation of the measured voltage.

In Eq. 2.3, the second multiplier is the ADC's LSB, defined in Eq. 2.1. For the ADS1247, the number of resolution bits N equals 24. However, because the ADS1247 PGA is supplied with a unipolar voltage, N should be set to 23 in Eq. 2.3.

From Eq. 2.2 and Eq. 2.3 it is easy to derive the formula for R_{MEAS}:

$$R_{MEAS} = \frac{R_{REF} * code}{2^{23} - 1 - code} \quad 2.4$$

As can be seen from Eq. 2.4, R_{MEAS} depends only on the reference resistance and not the reference voltage, which proves that this circuit is ratiometric.

The CE scheme has been studied very thoroughly since it is usually the first choice for precision temperature measurements. This scheme is mostly used with thermocouples or platinum resistance thermometers, which have a much smaller resistance range compared to thermistors. For example, the PT100 sensor used in [28] has a resistance of 60 Ω at -95° C and 140 Ω at 115° C, as opposed to 336.5 kΩ at -40° C and 5.3 kΩ at 40° C for the PS103J2 thermistor. The analysis, similar to the one described in 2.1.2, was performed using RT-characteristics of thermistors with the nominal resistances of 10 kΩ, 5 kΩ, 3 kΩ, and 2.25 kΩ.

To meet the power dissipation requirement of 10 µW, the excitation current should be smaller than 5.5 µA for the 10 kΩ, 7.6 µA for the 5kΩ, 9.9 µA for the 3 kΩ, and 11.3 µA for the 2.25 kΩ thermistor. However, the internal current source has the minimum excitation current of 50 µA, thus making it inapplicable for the CE scheme. Therefore, an external current source is required; for example, a Texas Instruments LM134 [29] can be used. The LM134 is a current source that can provide a set current as low as 2 µA. At minimum, the LM134 requires a resistor to set its current and a diode, if implementing the temperature compensation of the set current. Therefore, the drawback of using the LM134 is the increased components' count and, consequently, the higher cost and the larger size of the Logger.

The analysis revealed that implementing the CE scheme with a 10 kΩ thermistor would not be the most efficient option because of the thermistor's high resistance at low temperatures. The better choice would be thermistors with the nominal resistances of 5 kΩ, 3 kΩ, or 2.25 kΩ. The thermistor's lower resistance allows setting a higher excitation current, as well as having a smaller reference resistor. However, optimizing the scheme for these thermistors would make all existing probes (the installed and the manufactured ones) useless. Because of the above-mentioned reasons, the CE scheme was excluded from further consideration.

The bridge scheme has two drawbacks, which make it not suitable for the Logger. First, as discussed earlier in this section, the ADC's input voltage can have both negative and positive values, which requires a bipolar supply voltage for the ADC. This, in turn, requires a separate power supply system to be added to the Logger, as implemented in [30]. However, with the Logger's size and current

consumption requirements in mind, I was planning to design a less complicated power system consisting of a battery, a power switch and a voltage regulator.

Second, the measurement accuracy is affected by the fixed resistors' temperature coefficient. To make the error smaller, three very low temperature coefficient (TCR) resistors are required. These resistors are usually a bulk-metal through-hole type; they are much larger and much more expensive than surface-mount type resistors of the same value. For example, the Logger's reference resistor has the dimensions of 4.1 x 14.6 x 10.5 mm and it costs $17. The same value surface mount resistor has the dimensions of 0.6 x 0.3 mm and can costs as low as $0.1. If opting for the bridge scheme, it will compromise the size and the price requirements of the Logger.

The examination of three schemes showed that all of them could demonstrate similar performance with respect to temperature measurements if properly tuned-up. Therefore, other considerations predetermined the choice of the sensing scheme. Because the VE scheme required less components, thus reducing the Logger's price and size, I chose it for implementation in the Logger. Specifically, from four possible configurations I chose configuration #2 (Figure 2-4.(2)). My decision was justified entirely by the fact that, according to the datasheet, the ADS1247 supports single-ended input voltages. Two prototypes of the Logger (Board 1 and Board 2) were manufactured using the schematic and the printed circuit board designs presented in Appendix A and Appendix B respectively.

2.1.5 The Modification of the Logger's Sensing Circuit

Initial tests on Board 1 were performed where the thermistor was replaced with 18 resistors with resistances from 665 kΩ to 5 kΩ. The resistors were measured with a reference ohmmeter and the Logger. Equation 2.4 was used to calculate the resistance measured by the Logger. The measurements showed that there was a noticeable difference between the values measured by the Logger and the reference ohmmeter: up to 10 kΩ or 1.5% of the reading for the 665 kΩ resistor and up to 300 Ω or 6% of the reading for the 5 kΩ resistor. This section describes the steps taken to investigate the problem and changes made to the Logger's sensing circuit to achieve the required performance.

When investigating the problem, I studied the datasheet for the ADS1220, which has a PGA similar to the one used in the ADS1247. Unlike the datasheet for the ADS1247, the ADS1220 datasheet contains detailed information on the common-mode voltage requirements and provides very useful examples on the subject. When choosing the scheme for the sensing circuit, I did not verify that the circuit's common-mode voltage complied with these requirements. Therefore, I assumed that it could be the reason of the problem.

The following common-mode voltage requirements are specified in the ADS1247 datasheet:

$$V_{CM\_MIN} = AVSS + 0.1 + \frac{V_{IN} * Gain}{2} \quad 2.5$$

$$V_{CM\_MAX} = AVDD - 0.1 - \frac{V_{IN} * Gain}{2} \quad 2.6$$

where, V_{CM_MIN} is a minimum acceptable common-mode voltage, V_{CM_MAX} is a maximum acceptable common mode voltage, $AVSS$ is the analog most negative voltage, which is equal to 0 V, $AVDD$ is the analog most positive voltage, which is equal to 3.3 V, $Gain$ is the PGA's gain, and V_{IN} is the ADC's input voltage. V_{IN} is calculated as the difference between the positive and the negative input voltages:

$$V_{IN} = V_{INP} - V_{INN} \quad 2.7$$

The common-mode voltage is calculated using Eq. 2.8:

$$V_{CM} = \frac{V_{INP} + V_{INN}}{2} \quad 2.8$$

where V_{INP} and V_{INN} are the positive and the negative input voltage of the ADC.

The equations for positive V_{INP} and negative V_{INN} input voltages for all VE configurations are presented in Table 2-1.

Table 2-1. Positive and negative voltages for VE configurations

VE configuration	V_{INP}	V_{INN}
#1	$\frac{V_{REF}*(R_{THERM}+R_{MUX})}{R_{THERM}+R_{MUX}+R_{REF}}$ 2.9	0
#2	$\frac{V_{REF}*R_{REF}}{R_{THERM}+R_{MUX}+R_{REF}}$ 2.10	0
#3	V_{REF}	$\frac{V_{REF}*R_{REF}}{R_{THERM}+R_{MUX}+R_{REF}}$ 2.11
#4	V_{REF}	$\frac{V_{REF}*(R_{THERM}+R_{MUX})}{R_{THERM}+R_{MUX}+R_{REF}}$ 2.12

Figure 2-12 presents the plots of the PGA voltages for the VE configurations #1 through #4, as well as minimum and maximum acceptable common-mode voltages for these configurations, which were obtained from equations 2.5, 2.6, 2.8 and from Table 2-1. The gain setting of 1 was used in all cases. The figure displays data for the entire range of the thermistor resistance, which corresponds to the temperature range from -55° C to 76° C.

In Figure 2-12, the black line is the PGA's common-mode voltage V_{CM}, the black dashed line is the PGA's positive input voltage V_{INP} , the black dot-dashed line is the PGA's negative input voltage V_{INN} (for the configurations #1 and #2 this line aligns with the X-axis and, therefore, is invisible), the red

dot-dashed line is the maximum acceptable common-mode voltage V_{CM_MAX}, and the red dashed line is the minimum acceptable common-mode voltage V_{CM_MIN}.

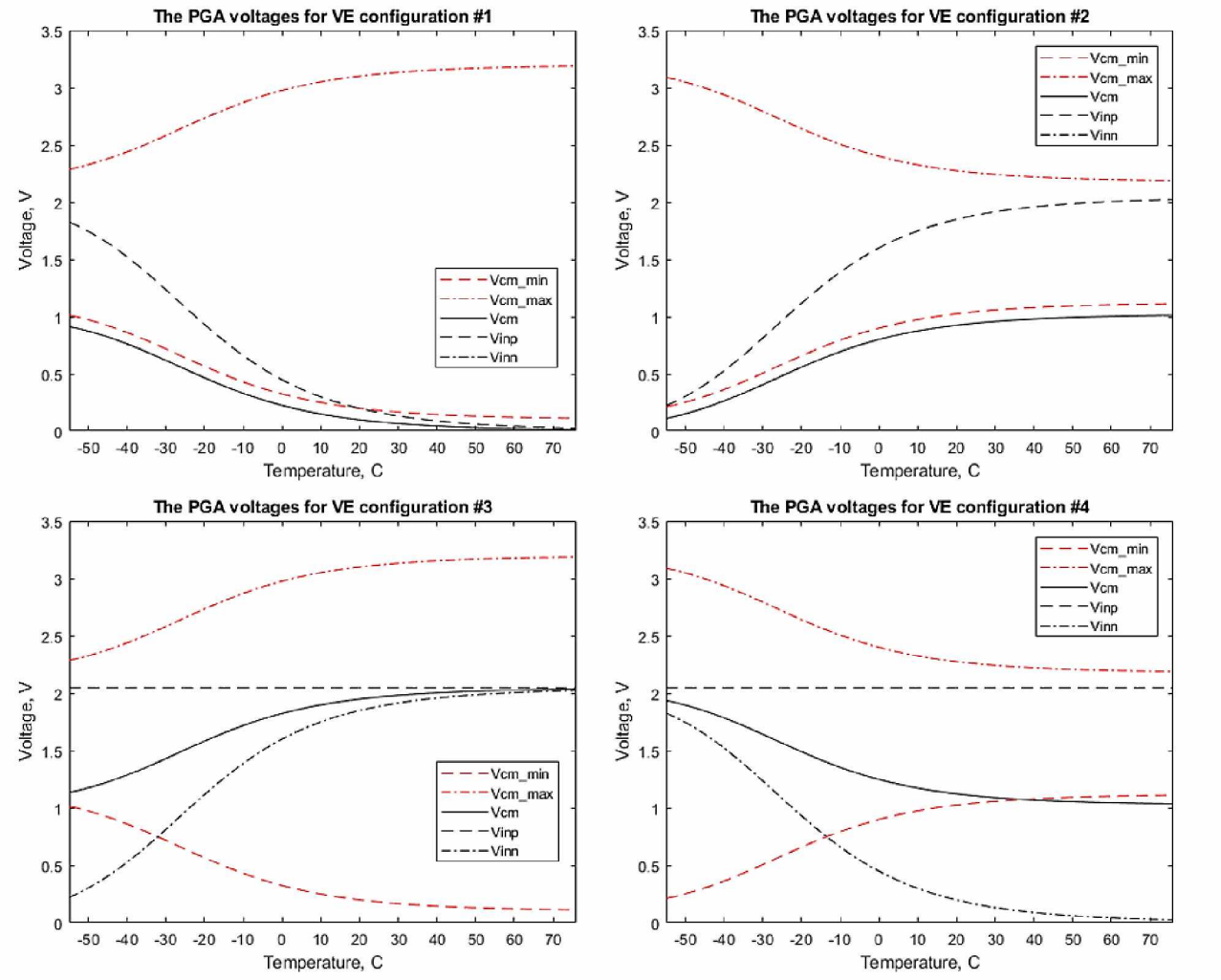

Figure 2-12. Common-mode voltages for the VE configurations 1 – 4

For the configurations #1 (Figure 2-4.(1)) and #2 (Figure 2-4.(2)), which is implemented in the Logger, the common-mode voltage V_{CM} curve lies below the minimum acceptable common-mode voltage V_{CM_MIN} curve, which indicates that these configurations violate the common-mode voltage requirement defined in Eq. 2.5.

For the configuration #3 (Figure 2-4.(3)), the common-mode voltage V_{CM} curve lies within the area bounded by the minimum V_{CM_MIN} and the maximum V_{CM_MAX} acceptable common-mode voltage curves, meaning that the V_{CM} satisfies requirements defined in both Eq. 2.5 and Eq. 2.6. For the configuration #4 (Figure 2-4.(4)), the common-mode voltage V_{CM} curve crosses the minimum acceptable

common-mode voltage V_{CM_MIN} curve at 37° C, making this configuration unusable for temperatures above 37° C.

Based on the results of the common-mode voltage calculations, I decided to change the Logger's schematic to the configuration #3 (Figure 2-4.(3)). The changes were implemented by soldering jumper wires on the printed circuit board of the first prototype (Board 1). This led to the removal of the analog multiplexer and the low-pass antialiasing filter from a signal path. Currently, Board 1 has a single hardwired channel #2 and no input low-pass filter.

The Logger's prototype with the modified schematic demonstrated considerably better results compared to the initial tests. For the 665 kΩ resistor, the error was 1.32 kΩ or 0.2% of the reading, and for the 5 kΩ resistor, the error was 1.29 or 0.03% of the reading. This proved the assumption that the common-mode voltage requirements violation was the cause of the problem. The second prototype of the Logger (Board 2) was modified differently – the analog multiplexer was kept, while the low-pass antialiasing filter was removed from the signal path. Currently, Board 2 has sixteen multiplexed channels and no input low-pass filter.

Meeting the common-mode voltage requirements of the PGA is an important design consideration, which was initially overlooked. Because of this, the temperature acquisition module failed to work properly in the beginning. As turned out, the ADS1247 can only measure single-ended voltages if its PGA is supplied with a bipolar voltage. In the Logger, however, it is supplied with a unipolar voltage. In this case, to measure single-ended voltages the ADC should have a PGA bypass feature, which is not available for the ADS1247. Future designs of the Logger will have the VE configuration #3 as the sensing circuit. The PGA gain will be set to 1.

2.1.6 Studying the VE Configuration #3

To evaluate the experimental results of the Logger, it was necessary to determine the expected temperature resolution of the VE configuration #3 (Figure 2-4.(3)). Even though existing probes use 10 kΩ thermistors, in the evaluation I used thermistors with three values of nominal resistance, namely 20 kΩ, 10 kΩ, and 5 kΩ. This was done to determine the best thermistor option for the Logger. It is important to note that both the 20 kΩ and the 5 kΩ thermistors meet the common-mode voltage requirements stated in equations 2.5 and 2.6 when used with the 116.667 kΩ reference resistor.

As discussed in 2.1.2, the reference resistance must be larger than 100 kΩ to ensure that the thermistor's dissipated power is smaller than 10 μW in the entire range of the thermistor's resistances. Additionally, as the reference resistance increases, the ADC's input voltage, as well as the input voltage

resolution, decreases. Therefore, it is preferable to have the reference resistance close to 100 kΩ. In the temperature range from -40° to 40° C, the 20 kΩ thermistor has resistances in the range 700 kΩ to 10 kΩ, the 10 kΩ thermistor has resistances in the range 337 kΩ to 5 kΩ, and the 5 kΩ thermistor has resistances in the range 170 kΩ to 2.5 kΩ; the RT-curves of the 5 kΩ, 10 kΩ, and 20 kΩ thermistors are presented in Figure 2-13.(left). For the configuration #3, the thermistor's power is maximal when the thermistor resistance is equal to the reference resistance. If choosing the reference resistance close to 100 kΩ, which falls within the resistance ranges for all thermistors, this value will be the best option for all thermistor types.

The expected temperature resolution can be calculated using Eq. 2.13:

$$dT_{EXP} = \frac{V_{NOISE}}{dV_{1°C}} \qquad 2.13$$

where, dT_{EXP} is the expected temperature resolution, V_{NOISE} is the input-referred noise voltage of the ADC's PGA, $dV_{1°C}$ is the input voltage resolution required to achieve 1° C temperature resolution.

Before the temperature resolution could be calculated, it was necessary to determine the unknown variables of Eq. 2.13, namely the input-referred noise voltage and the input voltage resolution.

Input-referred noise voltage values are provided in the ADS1247 data sheet. The noise values depend on the ADC's data rate, supply voltage, and the PGA gain. For example, with a 3.3 V supply voltage and the gain setting of 1, at a data rate of 5 SPS the peak-to-peak noise voltage is 14.24 µV. For the same conditions but at a data rate of 1000 SPS, the noise voltage is 388.28 µV. The input-referred noise determines the ADC's real LSB value, rather than the bit-resolution.

The input voltage resolution is calculated as the difference between the adjacent input voltage values. For the VE configuration #3, the input voltage is determined using equations 2.7 and 2.11. The following parameters were used for the calculations:

1. The voltage reference V_{REF} of 2.048 V for the ADS1247.
2. The multiplexer ON resistance R_{MUX} of 4.25 Ω for the ADG706.
3. The 20 kΩ, 10 kΩ, and 5 kΩ thermistors' RT-data as R_{THERM}.
4. The reference resistor R_{REF} of 116.667 kΩ.

The number of points in the calculated input voltage data was reduced by 20 times, which allowed to determine the input voltage resolution and, consequently, the expected temperature resolution with the temperature resolution of 1° C.

The ADC's input voltage resolution and, therefore, the anticipated temperature resolution with the 5 kΩ thermistor are worse in the temperature range from -20° C to 40° C than with the 10 kΩ or the

20 kΩ thermistors. Therefore, this type of thermistor has been removed from further consideration for the Logger and the evaluation results with the 5 kΩ thermistor are not presented.

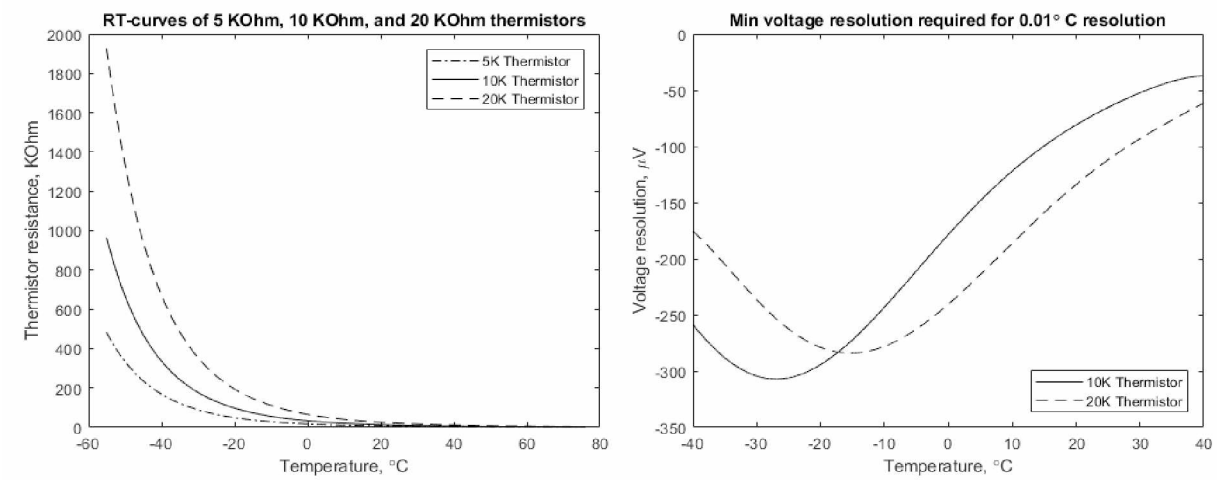

Figure 2-13. The RT-curves for 5 kΩ, 10 kΩ, and 20 kΩ thermistors (left) and the minimum voltage resolution for 0.01° C temperature resolution for 10 kΩ and 20 kΩ thermistors (right)

The curves of the minimum voltage resolution required for 0.01° C temperature resolution for both 10 kΩ and 20 kΩ thermistors are presented in Figure 2-13.(right). The expected temperature resolution for 10 kΩ and 20 kΩ thermistors is presented in Figure 2-14 (for all data rates) and Figure 2-15 (for data rates of 5 SPS and 10 SPS).

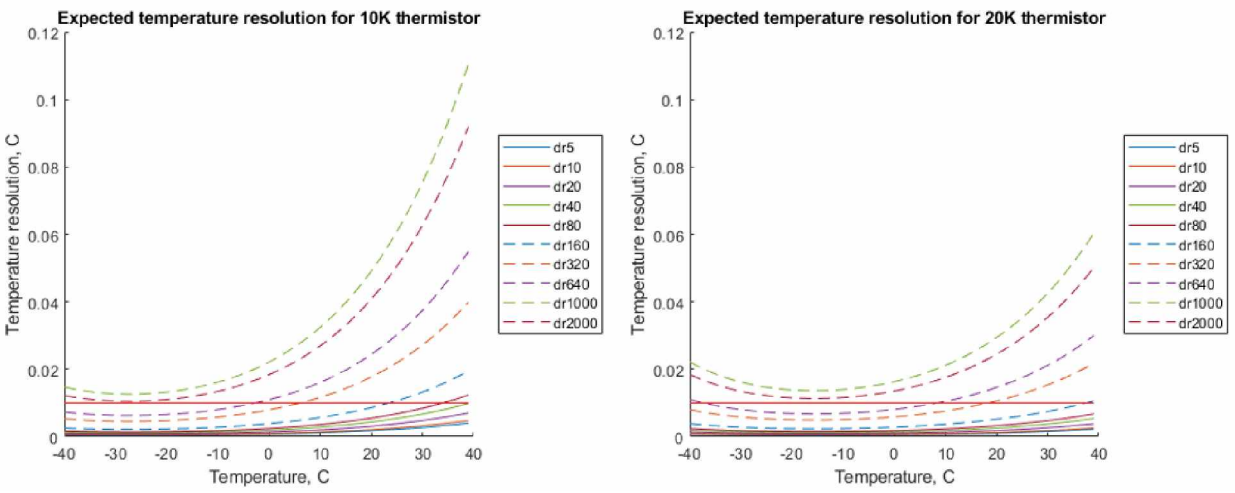

Figure 2-14. Expected temperature resolution for 10 kΩ (left) and 20 kΩ (right) thermistors for all data rates

If an application requires the temperature resolution to be better than 0.01° C, data rates not higher than 40 SPS for the 10 kΩ thermistor or 160 SPS for the 20 kΩ thermistor should be used. The 20 kΩ thermistor yields lower temperature resolution for temperatures above -15° C than the 10 kΩ thermistor. Also, since the input-referred noise voltage at 10 SPS is higher than at 5 SPS, temperature resolution at this data rate is slightly worse than at 5 SPS.

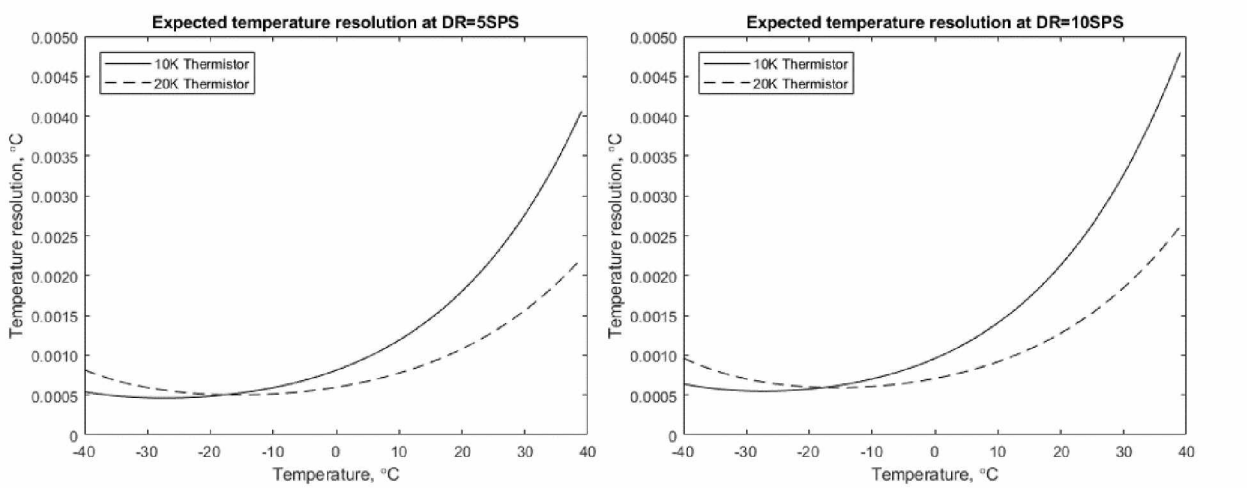

Figure 2-15. Expected temperature resolution for 10 kΩ (left) and 20 kΩ (right) thermistors for 5 SPS and 10 SPS

2.2 Design of the Power Supply Module

The power supply circuit of the Logger consists of the TPS2105 power multiplexer [31], the TPS78001 low dropout (LDO) voltage regulator [32], the TPS22929D load switch, which are all made by Texas Instruments, and a CR-P2 battery [33]. The schematic is shown in Figure A-2. The following section discusses the selection[1] of the power supply components and provides an estimation of the Logger's current drain and battery life.

2.2.1 Selecting the Power Supply Components

Before proceeding to the power supply components selection, it was necessary to determine the approximate current drain of the Logger. The estimation of the Logger's current drain, which was made using the components' datasheets, yielded the following results:

1. During the measurement, the current drain should be around 1 mA.
2. During the flash programming, the current drain should be around 15 mA.

[1] When selecting the components, only integrated circuits with package types allowing hand soldering (HTSSOP, SOIC, SOT) were considered.

3. During the microSD-card write, the current drain should be around 50 mA.
4. The above-mentioned currents will be of a pulsed-nature.

Because none of the power supply components depend on the load switch, it was selected first. A typical microSD-card drains around 250 – 400 µA in sleep mode. With the expected battery capacity of not more than 2500 mAh, this value is not acceptable for the application striving for a 2-year battery life. Therefore, the Texas Instruments TPS22929D load switch [34] was added to the design, which allows shutting down the microSD-card. At the time, the TPS22929D had the smallest typical quiescent current of 2 µA among all similar devices made by Texas Instruments; for most of the components the quiescent current was larger than 55 µA. On the other hand, its characteristics, such as ON resistance, input voltage, maximum load current, available quick discharge feature, were comparable to other load switches. Therefore, the TPS22929D was chosen for the Logger.

The power multiplexer allows switching the Logger's power source from the battery to the USB VBUS. The TPS2105 has a very small quiescent current, while other parameters, such as ON resistance of channels and maximum load current per channel, are comparable to other Texas Instruments power multiplexers. For this reason, it was selected for the Logger. The TPS2105 is controlled through an enable pin. When this pin is kept low, the second input is connected to the output. When the enable pin is asserted high, the TPS2105 switches to the first output. Because the battery is connected to the second input, the microcontroller is always powered from the battery in the beginning. When the USB cable is connected, the microcontroller senses the USB VBUS presence and commands the TPS2105 to switch to the first input, which is connected to the USB VBUS.

The TPS78001 voltage regulator was chosen because of its low quiescent current, low output noise, low dropout voltage, and acceptable maximum load current. The voltage regulator provides a regulated supply voltage of 3.3 V to all components of the Logger. The regulator's dropout voltage is presented in Table 2-2. Data are provided for the following conditions: V_{OUT} = 3.3 V, $V_{IN} = V_{OUT} + 0.5$ V, T_J = 25° C (junction temperature).

Table 2-2. The voltage regulator dropout voltage

Load current, mA	Dropout voltage, mV
< 1 mA	Negligible
15 mA	15
50 mA	40
150 mA	200

The supply voltage, the voltage drop across the power multiplexer ON resistance, and the dropout voltage of the voltage regulator set the limit on the battery's voltage, as expressed in Eq. 2.14:

$$V_{BATT\_MIN} = VCC + R_{MUX\_RON} * I_{LOAD\_MAX} + V_{DO}, \quad 2.14$$

where V_{BATT_MIN} is the minimal battery voltage, VCC is the supply voltage of 3.3 V, R_{MUX_RON} is the power multiplexer ON resistance, I_{LOAD_MAX} is the maximal load current, and V_{DO} is the dropout voltage of the voltage regulator. The TPS2105 power multiplexer ON resistance of the second input R_{MUX_RON}, where the battery is connected, is 1.3 Ω. Assuming the maximum load current I_{LOAD_MAX} of 50 mA, the voltage drop across the multiplexer will be 0.065 V. Taking into account the dropout voltage V_{DO} of 0.04 V (Table 2-2), the battery's voltage V_{BATT_MIN} must not be smaller than 3.405 V.

In general, only lithium ion batteries have been considered for the Logger because they perform better than alkaline batteries when operating at cold temperatures [35], [36]. In this section, the words "battery" or "cell" imply lithium ion battery or cell. Among the lithium batteries, a Lithium/Iron Disulfide battery with a nominal voltage of 1.5 V and an AA form-factor, for example Energizer L91 [35], or a Lithium/Manganese Dioxide battery with a nominal voltage of 6 V and CR-P2(223) or 2CR-5 form factors, for example Duracell Ultra 223 [33], were the two possible options.

Because the internal resistance of the lithium battery increases at low ambient temperatures [37], the battery's voltage decreases proportionally to the load current when operating in such conditions. A datasheet with discharge curves for different ambient temperatures, which have been measured at load currents that are close in value to the estimated currents of the Logger, would have helped to determine this voltage drop more precisely, which in turn would have facilitated the battery's selection. Such datasheets were not found for CR-P2 or 2CR-5 batteries, but many datasheets were found for the same chemistry batteries in another form-factor, namely CR-2, for example datasheets for Panasonic [38] and Hitachi-Maxell [39] batteries. And since a CR-P2 or a 2CR-5 battery consists of two CR-2 cells with a nominal voltage of 3 V connected in series, it is appropriate to use the CR-2 battery's characteristics for the power analysis.

If using an AA form-factor battery, three cells connected in series, which produce a nominal voltage of 4.5 V, will be required. At room temperature and for the constant load current of 10 mA, the cell voltage of Energizer L91 [35] quickly drops from initial 1.7 V to 1.5 V, then slowly decreases to 1.4 V, bringing the total battery voltage to 4.2 V. After that, the voltage drops abruptly to 0.8 V, indicating that the battery is dead (Figure 2-16.(left)).

At low temperatures down to -40° C and for constant load current of 25 mA, the capacity of the cell does not change. Reversing the reasoning in [40], this means that there is no increase in internal

resistance of the cell, and therefore, no voltage drop due to the operation in cold temperature. Therefore, the worst-case nominal voltage is 4.2 V, as determined at room temperature. Since 4.2 V is larger than the required minimal voltage V_{BATT_MIN} of 3.405 V, three AA cells is the acceptable option for the battery. The only downside of this choice is the overall size of 3 cells and inconvenience of their mounting on the Logger's board.

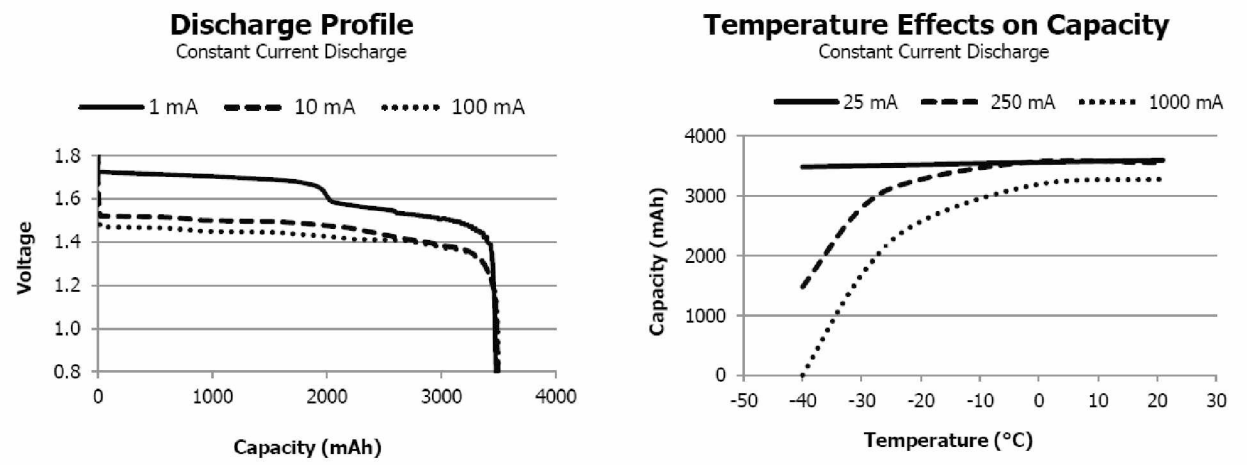

Figure 2-16. The Energizer L91 discharge profile (left) and temperature effects on capacity (right). Taken from [35]

For two CR-2 Hitachi-Maxell cells [39] (assuming it is a single CR-P2 battery), at room temperature and the constant load current of 30 mA, the cell voltage quickly drops from 3 V to 2.8 V or, equivalently, the battery voltage drops to 5.6 V. With time the cell voltage gradually decreases to 2.6 V or, equivalently, the battery voltage decreases to 5.2 V. After reaching this value, the cell voltage abruptly drops to 2 V, which indicates that the cell is dead. This process is shown in Figure 2-18.(right).

For this type of battery, the voltage drop associated with the cold temperature operation can be as large as 0.6 V per cell or 1.2 V for the battery. For example, in case of Panasonic battery [38], for the constant load current of 3 mA at -30° C, the cell's voltage drops by 0.3 V (Figure 2-17). In case of Hitachi-Maxwell battery [39], for the constant load current of 30 mA at -20° C, the cell's voltage drops by 0.6 V (Figure 2-18.(right)). Subtracting this voltage drop of 1.2 V from the nominal voltage of 6 V yields the difference of 4.8 V, which is larger than the required voltage V_{BATT_MIN} of 3.405 V. This makes the CR-P2 battery another acceptable option for the Logger's battery. The advantage of using the CR-P2 battery is its small size and ease of installation, compared to three AA cells. Mostly for this reason, the CR-P2 battery was chosen for the Logger.

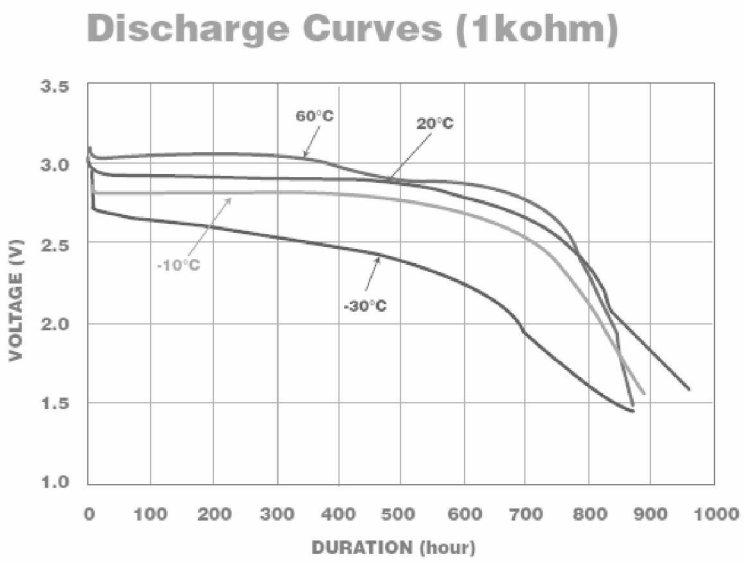

Figure 2-17. Discharge curves for Panasonic CR-2 battery. Taken from [38].

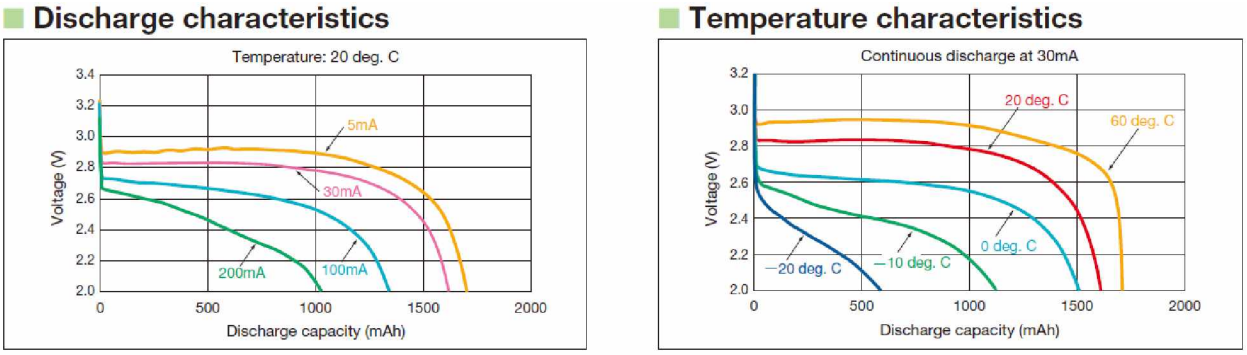

Figure 2-18. Discharge curves (left) and temperature effects on capacity (right) for Hitachi-Maxell CR-2 battery. Taken from [39]

2.2.2 Calculating the Expected Battery Life

After all components of the Logger were determined, it was possible to calculate the expected battery life. The required battery life is two years with the hourly measurement rate. The initial crude calculation of the battery life confirmed that the selected components would allow the Logger to meet the required 2-year lifespan.

Three main components determine the system's battery life: the amount of time spent in active and low-power modes (duty cycle), average current drain of each component, and the battery's capacity. Once the Logger's firmware was developed, the duration of certain operations was measured to provide more accurate battery life estimation. The timing was measured only for the repetitive operations, which primarily affect the average current drain of the Logger. Up to five measurements of

each operation were performed using one of the microcontroller's timers and were verified using the oscilloscope. The description and the timing of the measured operations is provided in Table 2-3.

Table 2-3. Timing of the Logger's repetitive operations

Parameter	Description of operation	Duration, ms
t1	Duration of tasks required before the measurement of 16 channels starts.	0.03345
t2	Time required to start the actual measurement of a single channel.	0.0061
t_{CONV}	Time required for a single conversion. Depends on data rate.	See Table 16 in [27]
t3	Time required to process the conversion result of a single channel.	0.02045
t4	Duration of tasks required before the next 16-channels measurement starts.	0.0611
t5	Time required to prepare data to be written to the microcontroller's flash	0.7591
t6	Time required to erase flash	6.78
t7	Time required to write flash	9.12
t8	Time required to write to microSD-card	725

Figure 2-19 shows one cycle of the Logger's operation. Numbers 1 – 4 indicate different stages of the cycle.

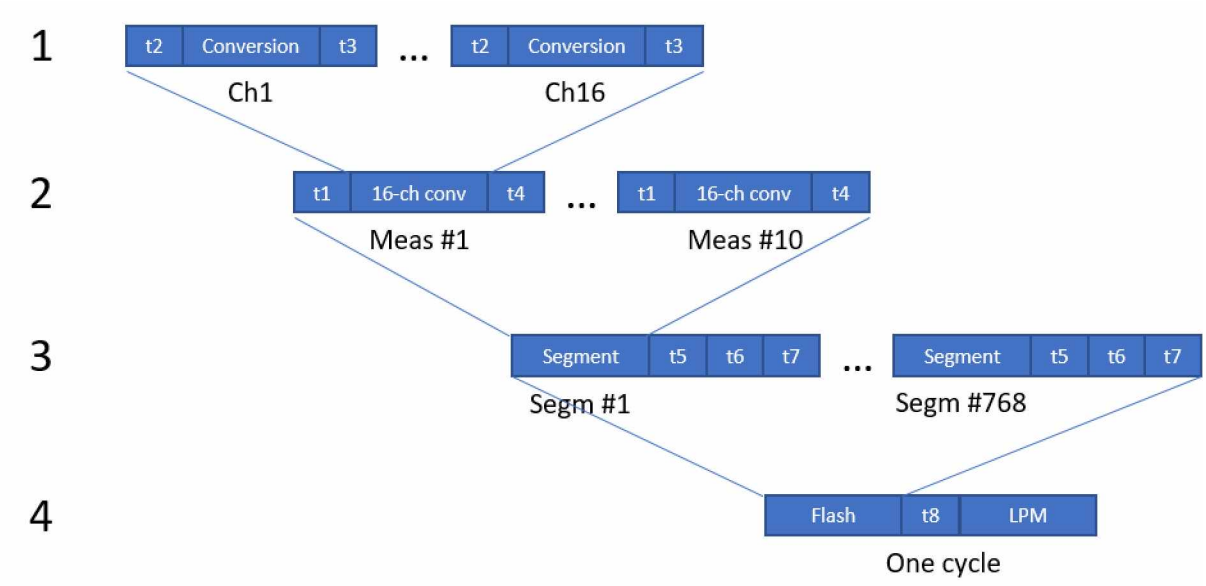

Figure 2-19. One cycle of the Logger's operation

Each measurement includes 16 individual channels' conversions. 10 measurements fill up the microcontroller's flash segment. The 384 KB of flash memory, reserved for data storage, contain 768 segments. Once the 384 KB of flash is full, data are written to micro-SD card. The duration of a low-power mode (labeled as "LPM" in Figure 2-19) is a cumulative time the Logger spends in a low-power mode. In fact, the Logger sleeps in between the measurements, which are separated by the

measurement intervals. The Logger is active during the first part of the measurement interval and sleeps during the second part of this interval. Four tasks can be performed individually or simultaneously during one measurement interval: measurement, calibration, writing to flash, and writing to microSD-card. For each data rate, the minimal measurement intervals were determined by measuring time required to simultaneously perform calibration, measurement, writing to flash, and writing to microSD-card. The minimal measurement intervals are:

1. 12 s for a data rate of 5 SPS.
2. 6 s for a data rate of 10 SPS.
3. 4 s for data rates of 20 SPS and 40 SPS.
4. 2 s for data rates of 80 SPS, 160 SPS, 320 SPS, 640 SPS, 1000 SPS, and 2000 SPS.

The next component of the battery life estimation is the average current drain. All components constantly draw some amount of current from the power supply, which is required for their operation. The sum of the components' currents determines the minimal current drain. Table 2-4 summarizes the current drain for most of the Logger's components, excluding the ADS1247, the MSP430, and the microSD-card. Current drain of the ADS1247 and the MSP430 is discussed further in this section. The typical and maximal current drains of a Swissbit u200 microSD-card [41], which is used for the Logger, are 40 mA (write), 30 mA (read), and 50 mA (write/read). The presented current drains were measured by the manufacturer at room temperature of 25° C, unless otherwise noted.

Table 2-4. Current drain of the Logger's components

Component	Test conditions	Current, µA	
		Typical	Maximum
Analog MUX ADG706	Supply voltage 5.5 V	0.001	1.0
Radio	Sleep mode. Supply voltage 3 V	0.2	1.0
Power MUX TPS2105	Typical current at T_J = 25° C Maximal current at -40° C $\leq T_J \leq$ 125° C	0.75	 1.5
Voltage regulator TPS78001	V_{IN} = 5.5 V, V_{OUT} = 3.3 V Typical current: at $I_{OUT} \leq$ 1 mA (in a low-power mode) at I_{OUT} = 1 mA (during the measurement) at I_{OUT} = 15 mA (during flash write) Maximal current: at I_{OUT} = 50 mA during write to microSD-card	 0.5 0.5 2.0 	 3.0
Load MUX TPS22929D	microSD-card is turned off. Leakage current, V_{ON}= GND, V_{OUT} = 0 V, V_{IN} = 3.6 V	0.2	7.0
Load MUX TPS22929D	microSD-card is turned on. Quiescent current, V_{OUT} = V_{ON} = V_{IN} = 3.6 V	2.0	7.0
ESD diodes TPD2E001	Supply voltage 5 V, ambient temperature -40° to 85° C	N/A	0.001

The ADS1247 current drain depends on the data rate [27]. When internal reference voltage source is turned on, which happens only during the conversion process, the ADS1247 drains additional current of 180 µA. The MSP430 current drain at different operational modes is presented in Table 2-6.

Table 2-5. The ADS1247 current drain at supply voltage of 3.3 V

Data rate	Analog current drain, µA	Digital current drain, µA
5	225	210
10	225	210
20	225	210
40	325	220
80	325	220
160	325	220
320	400	235
640	400	235
1000	400	235
2000	500	260
LPM	0.1	0.2

Table 2-6. The MSP430 current drain

The MSP430 operational mode	Typical current drain	Maximum current drain
Active mode, program execution from flash, supply voltage 3.3 V, VCORE = 3, clock frequency 20 MHz	7.4 mA	N/A
LPM3, current for the real-time clock sourced from a 32786 Hz crystal is included, supply voltage 3.3V, VCORE = 3	2.6 µA	4.2 µA
During flash program	3.0 mA	5.0 mA
During flash erase	6.0 mA	15.0 mA

Using data from Table 2-4, the minimal worst-case current is determined as the sum of maximal currents[2] of the Logger's components, which is equal to $I_{MIN} = 1 + 1 + 1.5 + 0.5 + 7 + 0.001 = 11.001$ µA.

The following equations were used to determine the product of the current and the duration of certain operations (Table 2-3), which occur during one cycle (Figure 2-19):

$$IT_1 = N_{CH} * N_{MS} * N_{SF} * ((I_{MSP\_AM} + I_{ADC\_A\_LPM} + I_{ADC\_D\_LPM}) * t2 + (I_{MSP\_LPM} + I_{ADC\_A\_AM} + I_{ADC\_D\_AM}) * t_{CONV} + (I_{MSP\_AM} + I_{ADC\_A\_LPM} + I_{ADC\_D\_AM}) * t3) \quad (2.15)$$

[2] Except the TPS78001 voltage regulator, for which the current at I_{OUT} = 1 mA was taken, because the voltage regulator spends most of the time in this mode.

$$IT_2 = N_{MS} * N_{SF} * ((I_{MSP\_AM} + I_{ADC\_A\_LPM} + I_{ADC\_D\_LPM}) * t1 + (I_{MSP\_AM} + I_{ADC\_A\_LPM} + I_{ADC\_D\_LPM}) * t4) + IT_1 \quad 2.16$$

$$IT_3 = N_{SF} * ((I_{MSP\_AM} + I_{ADC\_A\_LPM} + I_{ADC\_D\_LPM}) * t5 + (I_{MSP\_FE} + I_{ADC\_A\_LPM} + I_{ADC\_D\_LPM}) * t6 + (I_{MSP\_FW} + I_{ADC\_A\_LPM} + I_{ADC\_D\_LPM}) * t7) + IT_2 \quad 2.17$$

$$IT_4 = (I_{MSP\_AM} + I_{SD\_W} + I_{ADC\_A\_LPM} + I_{ADC\_D\_LPM}) * t8 + IT_3 \quad 2.18$$

In equations 2.15 – 2.18, $IT_1 - IT_4$ are the products of the current and the duration of operations for the stages 1 through 4 (Figure 2-19), N_{CH} is the number of channels, which is equal to 16, N_{MS} is the number of measurements of all channels per segment of 512 bytes, which is equal to 10, N_{SF} is the number of segments in 384 KB of flash memory reserved for data storage, which is equal to 768, I_{MSP_AM} is the typical current drain of the MSP430 in active mode, I_{MSP_LPM} is the MSP430 worst-case current drain in a low-power mode, I_{MSP_FE} and I_{MSP_FW} are the worst-case MSP430 currents during flash erase and write respectively, all defined in Table 2-6, $I_{ADC\_A\_LPM}$, $I_{ADC\_D\_LPM}$, $I_{ADC\_A\_AM}$, $I_{ADC\_D\_AM}$ are the ADS1247 analog and digital currents in a low-power and active modes, as defined in Table 2-5, I_{SD_W} is the microSD-card worst-case current during write operation as defined in Table 2-4, $t1 - t8$ and t_{CONV} are timing parameters defined in Table 2-3.

Since there is no indication in the ADS1247 datasheet whether digital circuit is operational during the conversion, it was assumed that both the analog and the digital circuits are active. The ADS1247 datasheet clearly states that during the conversion result transfer, the analog circuit is in a low-power mode. Additionally, the MSP430 stays in a low-power mode during the conversion.

The total time of one cycle t_{TOT} is determined using Eq. 2.19 below:

$$t_{TOT} = N_{MS} * N_{SF} * t_{MEAS} \quad 2.19$$

where t_{MEAS} is the measurement interval of the Logger.

The time spent in active mode can be determined as:

$$t_{AM} = N_{CH} * N_{MS} * N_{SF} * (t2 + t_{CONV} + t3) + N_{MS} * N_{SF} * (t1 + t4) + N_{SF} * (t5 + t6 + t7) + t8 \quad 2.20$$

Knowing the total time of one cycle t_{TOT} and the time spent in active mode t_{AM}, it is easy to determine the time spent in a low-power mode t_{LPM} as the difference between the two.

Using equations 2.15 – 2.18 and 2.20, the Logger's average current in active mode I_{AM}, the average current in a low-power mode I_{LPM}, and the overall average current can be determined as:

$$I_{AM} = \frac{IT_4}{t_{TOT}} \quad 2.21$$

$$I_{LPM} = \frac{(I_{MSP\_LPM} + I_{ADC\_A\_LPM} + I_{ADC\_D\_LPM}) * t_{LPM}}{t_{TOT}} \quad 2.22$$

$$I = I_{AM} + I_{LPM} + I_{MIN} \quad 2.23$$

In Eq. 2.23, I_{MIN} is the minimal current drain of 11.001 µA, which was determined earlier in this section.

The last component of the battery life estimation is the battery's capacity. The capacity depends on the load current, ambient temperature, and cutoff voltage.

The cutoff voltage determines the battery's voltage below which the battery is considered dead. This parameter is chosen by the power supply system designer. For the CR-P2 battery, the cutoff voltage was chosen to be 5 V. This value maintains the minimum required battery voltage V_{BATT_MIN} of 3.405 V determined using Eq. 2.14 plus a possible voltage drop of 1.2 V due to operation in cold temperature.

The battery's capacity greatly depends on ambient temperature [37]. The Logger is intended to be installed in the permafrost thermistor probe. A typical site will have the probe buried into the ground with the Logger near the ground surface. Ground surface temperatures and mean annual ground surface temperatures (MAGST) for GIPL sites deployed near Fairbanks, Alaska vary within the range from -20° to 40° C and -3° C to slightly below 0° C respectively [1]. As discussed in [37], chemical processes that lead to increase in internal resistance of a battery are reversible. This means that the battery's voltage and, therefore, the capacity will fluctuate with the temperature change assuming the same load current. Therefore, MAGST was used to obtain the average battery's capacity. The average capacity was determined from Figure 2-18.(right) as the value at the intersection of the discharge curve measured at 0° C (this temperature lies within the MAGST range) and the cutoff voltage of 2.5 V (it corresponds to the battery voltage of 5 V), which is equal to 1250 mAh. Since the battery consists of two CR-2 cells connected in series, its capacity is equal to the capacity of one cell, or 1250 mAh.

Using equations 2.15 – 2.23 , data from Table 2-5 and Table 2-6, the timing parameters from Table 2-3, and the average battery's capacity, the average currents for the cases of the minimal and 1-hour measurement intervals were calculated and are shown in Figure 2-20.(left) and Figure 2-21.(left). Battery life was calculated as battery capacity in Coulombs divided by the average current (Figure 2-20.(right), Figure 2-21.(right)). Matlab was used to perform both calculations.

The decrease in average current and, consequently, the increase in battery life is explained by the fact that the conversion time is much smaller at higher data rates than at lower data rates; the

higher current drain at higher data rates does not change this behavior in most cases except for the case with data rates of 40 SPS and 80 SPS.

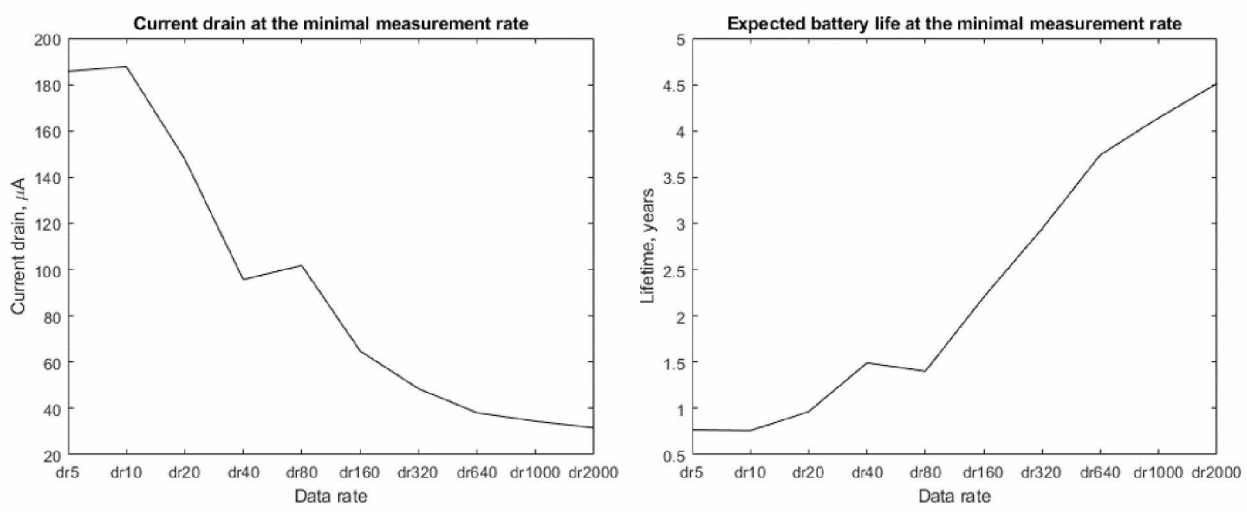

Figure 2-20. Average current (left) and battery life (right) for the minimal measurement interval

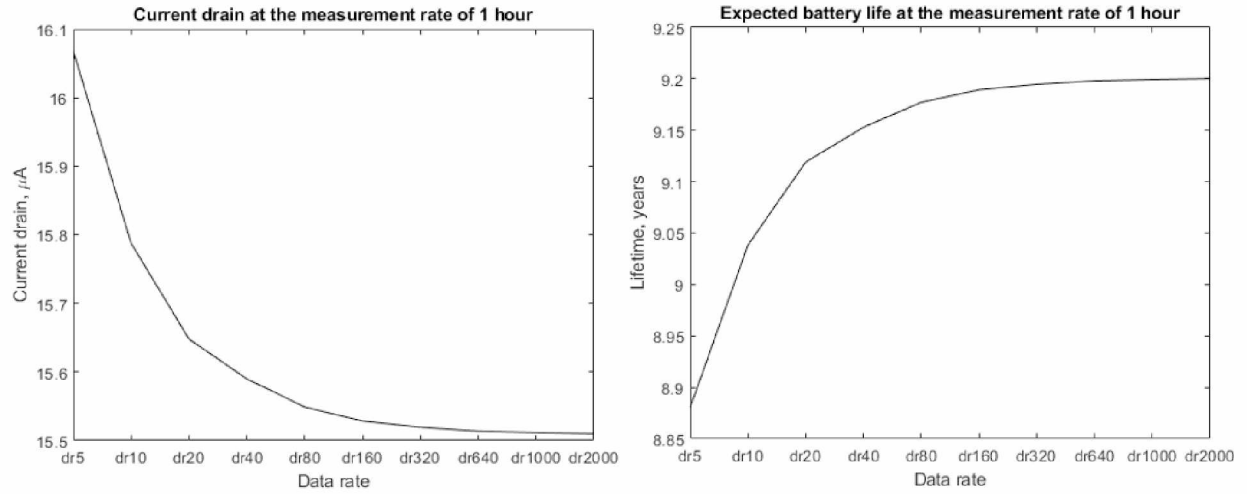

Figure 2-21. Average current (left) and battery life (right) for 1-hour measurement interval

2.2.3 The Voltage Regulator Efficiency and Power Dissipation

The voltage regulator efficiency can be calculated using Eq. 2.24 [42]:

$$Efficiency = \frac{I_{OUT} * V_{OUT}}{(I_{OUT} + I_Q) * V_{IN}} * 100 \qquad 2.24$$

where I_{OUT} is the output current equal to the average current determined above, V_{OUT} is the output voltage equal to 3.3 V for the TPS78001, V_{IN} is the input voltage approximately equal to 5.6 V for the

CR-P2 battery as discussed earlier in this section, and I_Q is the quiescent current of the TPS78001, which is equal to 0.5 µA as measured by the manufacturer at the following conditions: T_J=25 °C, V_{IN} from 3.8 to 5.6 V, V_{OUT} = 3.3 V, I_{OUT} = 1 mA.

The voltage regulator efficiency depends on the difference between the input and output voltages and the quiescent current; it does not depend much on the output current. Therefore, for all data rates efficiency is the same and is equal to 58.64 %.

The maximal allowable dissipated power P_{D_MAX} of the voltage regulator can be calculated using Eq. 2.25:

_____

where T_{J_MAX} is the maximal allowable junction temperature equal to 125° C for the TPS78001, T_A is the maximal ambient temperature equal to 60° C for the Logger, and $R_{\theta JA}$ is the junction-to-ambient thermal resistance equal to 193 C°/W for the TPS78001 with 5-pin package. The calculated maximal dissipated power is 336.79 mW.

The maximal dissipated power P_D of the voltage regulator can be found using Eq. 2.26:

Using the same values as in Eq. 2.24 and I_{OUT_MAX}= 50 mA (the maximal current of the microSD-card used for the Logger), the maximal dissipated power is 115 mW, which is smaller than the maximal allowable dissipated power of 336.79 mW. All other components of the Logger stay within the allowable limits on the maximal dissipated power (for the TPS22929D, the ADG706, the ADS1247, and the MSP430) or power rating (for the TPS2105) as well.

2.2.4 Other Supporting Hardware

The Texas Instruments MSP430F5659 microcontroller (MSP430) [15] is used for the Logger (Figure A-2). It is a very low power device with a large set of features. It has 512 KB of flash, six universal serial communication interface (USCI) modules, 74 digital inputs/output ports, a real-time clock timer RTC B, an embedded USB module, etc. The microcontroller uses three USCI modules UCA1, UCB1, and UCA2 to communicate with peripherals of the Logger – the radio, the microSD-card and the ADC respectively. All three USCI modules are set to operate in SPI mode. The MSP430 uses general purpose input/output ports to control other peripherals – the analog multiplexer, the voltage regulator, and power switches.

The Logger's memory consists of the MSP430 flash and a microSD-card (Figure A-2). From 512 KB flash, 384 KB are used to store measurement data, time stamps, and cyclic redundancy check (CRC) information. The microSD-cards' electrical interface is designed to support standard capacity cards of up to 2 GB.

The MSP430 can be programmed through the JTAG interface or through the USB. The JTAG is the primary interface for the firmware upload and debug. Programming through the USB allows a user to update the microcontroller's firmware in the field without special devices like flash emulation tool (FET). Once the production version of the Logger is ready, the firmware will be uploaded over the USB and the JTAG interface will be completely removed to reduce the size and cost of the device.

The MSP430 embedded USB module is used for communication with a host PC. Once the Logger is enclosed in a case, the only entry point for the ESD will be the USB interface. To protect the device, the Texas Instruments TPD2E001 [43] ESD protection diode array was incorporated into the USB interface design (Figure A-2).

The Anaren A110LR09A radio [44] is used to provide wireless capabilities for the Logger (Figure A-2). It is a 900 MHz radio with programmable output power, data rate, and modulation scheme, which is controlled over the SPI interface.

2.3 PCB Design and the Final Product

For a printed circuit board (PCB) implementation, I chose a 4-layer substrate. The top and the bottom layers are used for signal routing, the second layer from the top is a ground plane, and the third layer from the top is a power supply plane. When designing the PCB, I followed the best practices of signal and power integrity described in [45], [46], [47]. The designed board is presented in Appendix B; the manufactured prototype (Board 2) is shown in Figure 2-22.

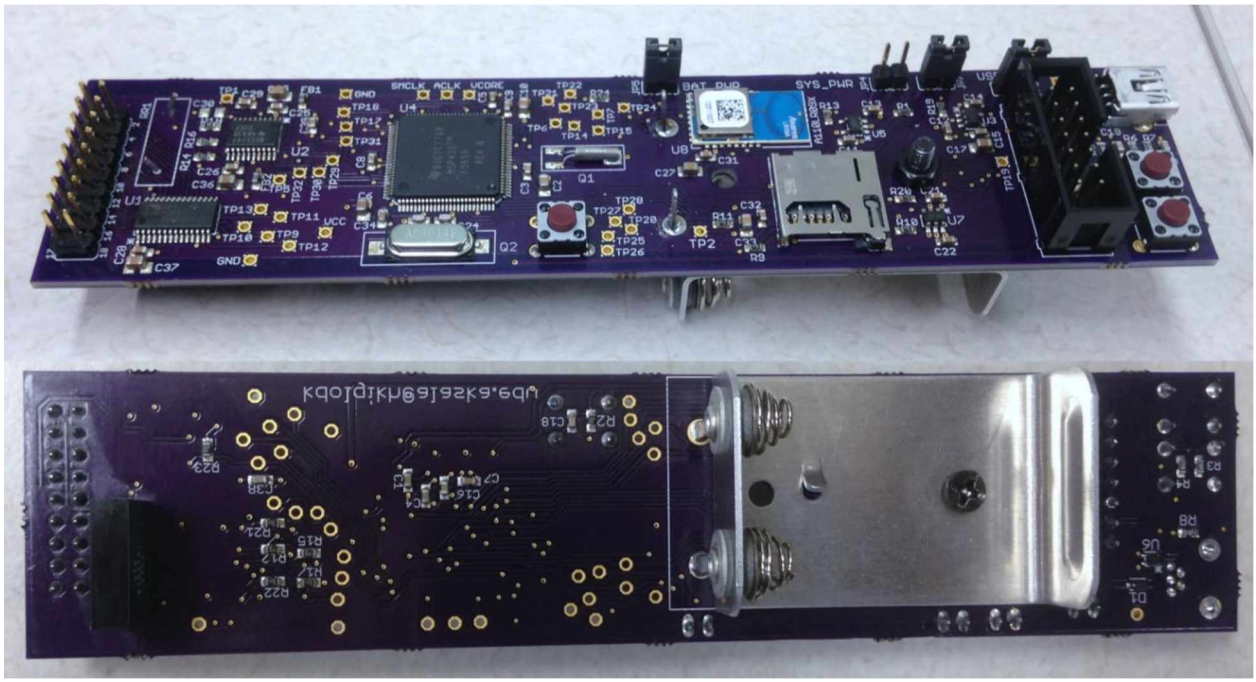

Figure 2-22. The prototype of the Logger (Board 2)

Chapter 3 Firmware Architecture

The development of the firmware for the Logger went through several stages. First, a simple code to test the boards' operation was created. It included setting up clock system, enabling crystals, and configuring ports. Next, peripheral drivers were created and tested. Finally, the code to fulfill the Logger's functionality was developed. This Chapter's organization, however, does not follow this sequence: first, the general overview of the main function is presented; second, operation during the disconnected and enumerated states of the main function is described; finally, the peripherals' drivers are discussed. This organization is reflected in the block diagram presented in Figure 3-1.The code itself is presented in Appendix F.

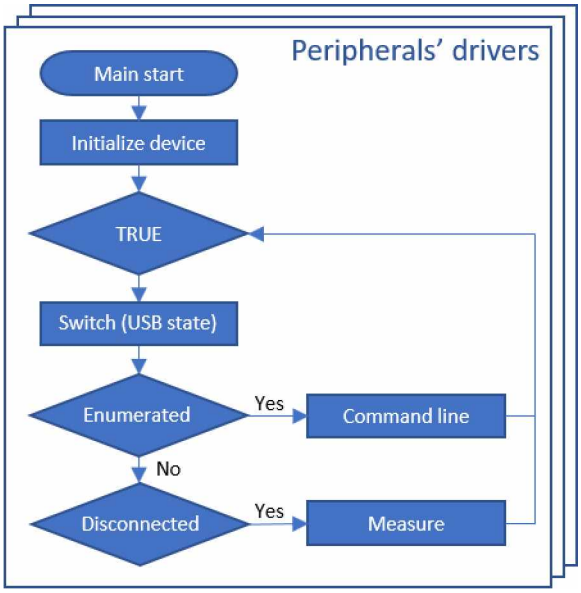

Figure 3-1. The firmware block diagram

3.1 Main program

The main program's flowchart is presented in Figure 3-2. The main program starts with the initialization stage, which will be described in detail in the next subsection. After initialization, the program proceeds to the infinite loop, which contains a state-machine built around the USB interface states: (#1) the USB is enumerated, (#2) the USB is disconnected, (#3) the USB is connected but not enumerated, or the USB is connected and suspended, or the USB is connected but not enumerated or suspended, (#4) enumeration is in progress. Another state-machine that governs the program's

operation when the Logger is disconnected from the host PC (state #2) is based on the device state, which can be either “Measuring”, “Memory full”, “Malfunction”, or “USB Connected.”

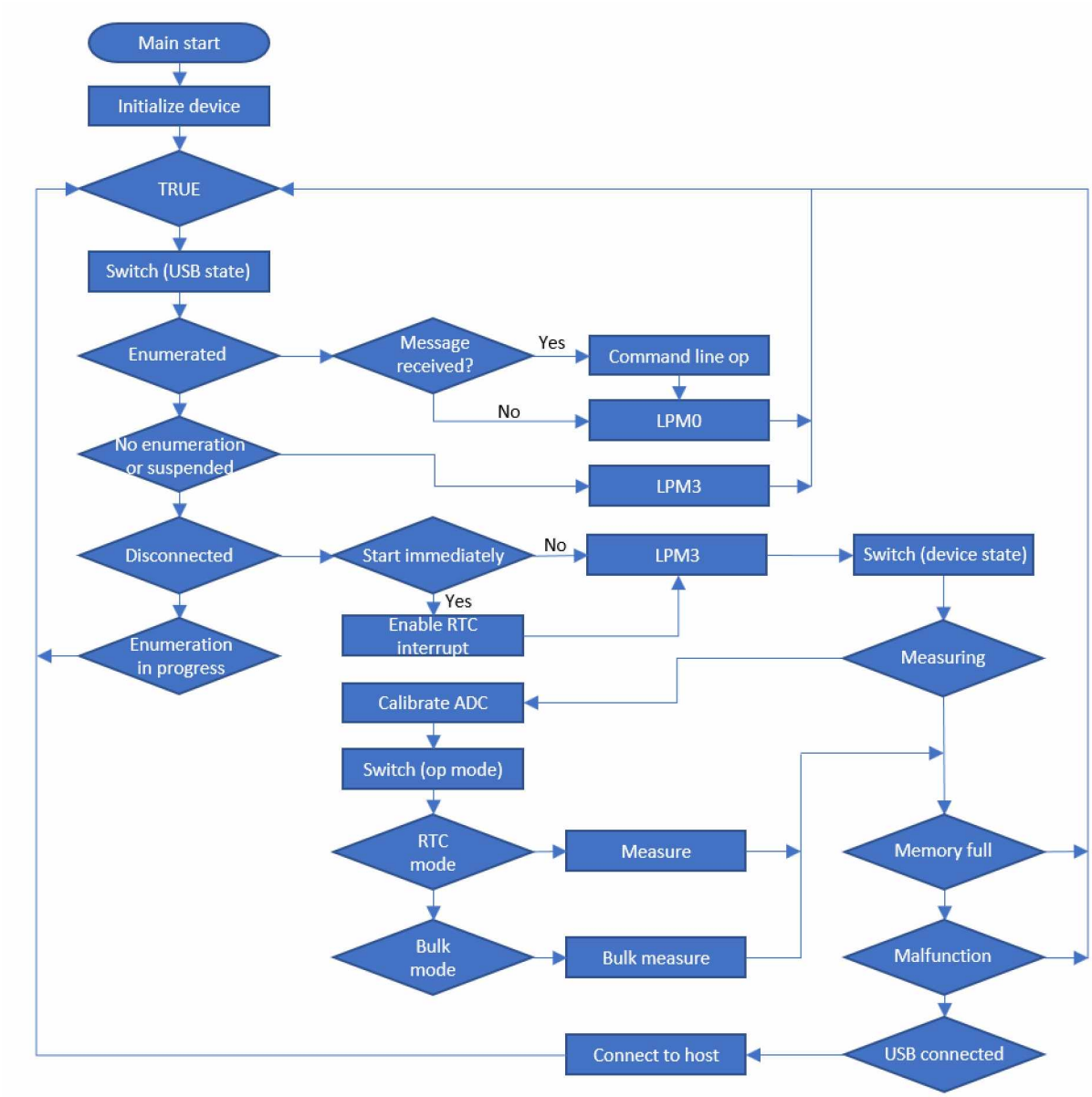

Figure 3-2. The main program’s flowchart

When the Logger is connected to the host PC and enumerated (state #1), a user can use a command line interface, which runs within the MATLAB’s workspace, to setup the Logger’s parameters for the future measurement session. Details on the command line interface are presented in Chapter 4.

When the user disconnects the Logger from the host PC, the USB state changes to disconnected (state #2) and the device state changes to “Measuring”. Next, if the user has set the “start immediately” parameter, the microcontroller enables the RTC B interrupt, and enters the LPM3. If the “start

immediately" parameter has not been set, the Logger proceeds to the LPM3 and waits in this mode until it has been connected to the host PC again. The RTC B interrupt wakes up the Logger at exactly 2-second boundary. This synchronization is required by the measurement functions, which are discussed in 3.1.2, and the function that generates a time stamp.

The Logger has two measurement modes: "RTC mode" and "Bulk mode." In "RTC mode," a 16-channel measurement is repeated at the user selectable rate, which is maintained by the real-time clock timer. The minimum measurement rate cannot be less than 2 seconds, while the maximum rate is not restricted. In "Bulk mode," each channel's measurement is repeated non-stop until the predefined number of segments (so-called "num_bulk" number) is reached, then the program moves to the next channel.

The minimum set of parameters for the "RTC mode" includes setting the measurement mode, the real-time clock, the measurement rate, the data rate, and the measurement start. For the "Bulk mode," the measurement mode, the number of segments ("num_bulk"), the data rate, and the measurement start are required to be set. Currently, the only implemented measurement start option is the "immediate start." Once these parameters are set through the command line interface, the measurement session can be started.

Based on the configured measurement mode, the microcontroller calls the measure_all_channels_store_flash_sdc() function (for the "RTC mode") or the measure_bulk_all_channels_store_flash() function (for the "Bulk mode"). The measurement session continues until the memory is full, the Logger has encountered an error during its operation, or the Logger is connected to the host PC. If the Logger's memory becomes full, the device state changes from "Measuring" to "Memory full." If there is an error during the communication with the ADC, the device state changes from "Measuring" to "Malfunction." In both cases, the microcontroller then enters the LPM3. If the Logger is connected to the host PC while being in any state, the device state changes to "USB connected" and the usb_connect2host() function is called, which requests the host PC to start the enumeration process.

If the program enters the USB state #3, the microcontroller goes to the LPM3 mode and stays in this mode until the host PC decides to enumerate it. In the USB state #4, the USB API performs enumeration, so the microcontroller should stay in active mode and wait until this process has finished. Therefore, the program performs no operation and immediately re-evaluates the USB state.

3.1.1 Initialization

After reset, the microcontroller goes through the initialization. Several functions are used during the initialization: device_init(), Radio_Init(), and USB_setup().

The device_init() function contains several sub-functions to initialize the following objects: unified clock system (UCS), timer A2 (TA2), SPI interfaces to the ADC, the microSD-card, and the radio, unused ports, the user button port, the analog multiplexer control ports, and the supply voltage supervisor (SVS) and monitor (SVM).

The MSP430 master clock (MCLK) is set to 20 MHz. To support this setting, the core voltage level is increased to 1.9 V, which corresponds to core level 3. The MCLK is sourced from the digitally-controlled oscillator (DCO) that is stabilized by the FLL circuit. The FLL circuit is sourced from the 32768 Hz crystal Q1.

TA2 is used for the Sleep_Timer_Fast() function, which allows the microcontroller to be in a low power mode for a predefined number of clock cycles. In case of TA2, this clock is the auxiliary clock (ACLK) sourced from the Q1 crystal. The crystal's frequency of 32768 Hz is divided by eight to provide the TA2 clock of 4096 Hz. Therefore, one clock cycle takes 0.244 ms and this is the minimum amount of time the microcontroller can spend in the low power mode when using this function.

The radio-to-radio communication protocol is not implemented in the current version of the firmware. Currently, only the SPI protocol to control the radio is implemented. Therefore, the Radio_Init() function's purpose is to put the radio into the low power mode by sending the appropriate command over the SPI interface.

Disabling the SVS and SVM modules reduces the microcontroller's power consumption and shortens the LPM-to-AM transition time from 165 µs to 3 µs. Additionally, all unused ports are set to output direction, ground potential ("Low"), which helps reducing power as well. Only two low power modes are used for the microcontroller – the LPM0 and the LPM3. The LPM0 is used when the Logger is connected to the host over the USB, and the LPM3 is used when the Logger is disconnected from the host. In low power mode 4 (LPM4), the Q1 crystal, which is required for the real-time clock operation, is disabled. Therefore, the Logger could not benefit from the LPM4's ultra-low power consumption.

3.1.2 Functions to Measure Resistance

3.1.2.1 Measuring in the "RTC Mode"

The measure_all_channels_store_flash _sdc() function performs the measurement in the "RTC mode." The main parameter that controls the function's operation is the number of segments that have been written into the Logger's memory. Each segment consists of: (1) the time stamp (5 bytes) and the time stamp CRC (2 bytes), (2) 10 blocks of 16-channel measurements (48 bytes) plus the CRC (2 bytes), (3) five spare bytes (Figure 3-3). This yields a total of 512 bytes. Because the time stamp is generated only for the segment and not for each individual measurement, a considerable amount of memory is saved for useful data. Time stamps for each individual measurement are generated at the host PC's software, which will be discussed in the next Chapter.

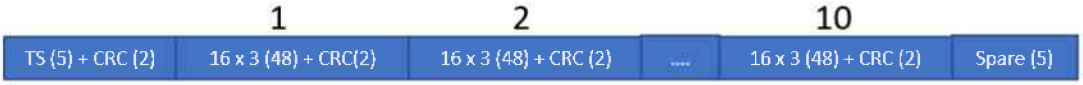

Figure 3-3. Data segment structure

The flowchart of the measure_all_channels_store_flash _sdc() function, which performs measurements in the "RTC mode," is shown in Figure 3-4. The function keeps track of the segments written to the Logger's memory. Until the segment is full, it is stored in an array in the microcontroller's RAM storage. Once the segment is filled with data, it is written to the part of the microcontroller's flash memory (384 KB) reserved for data storage, which is called "flash storage." When flash storage is full, data from it is written to microSD-card. This process repeats until the microSD-card is full. Then, the function writes data to the flash storage for the last time. When all storage locations are full, the function declares the Logger's memory full and returns to the main loop.

An important note is that the function supports the USB hot-plugging functionality, which allows connecting the Logger to the host PC at any time without the fear of losing data.

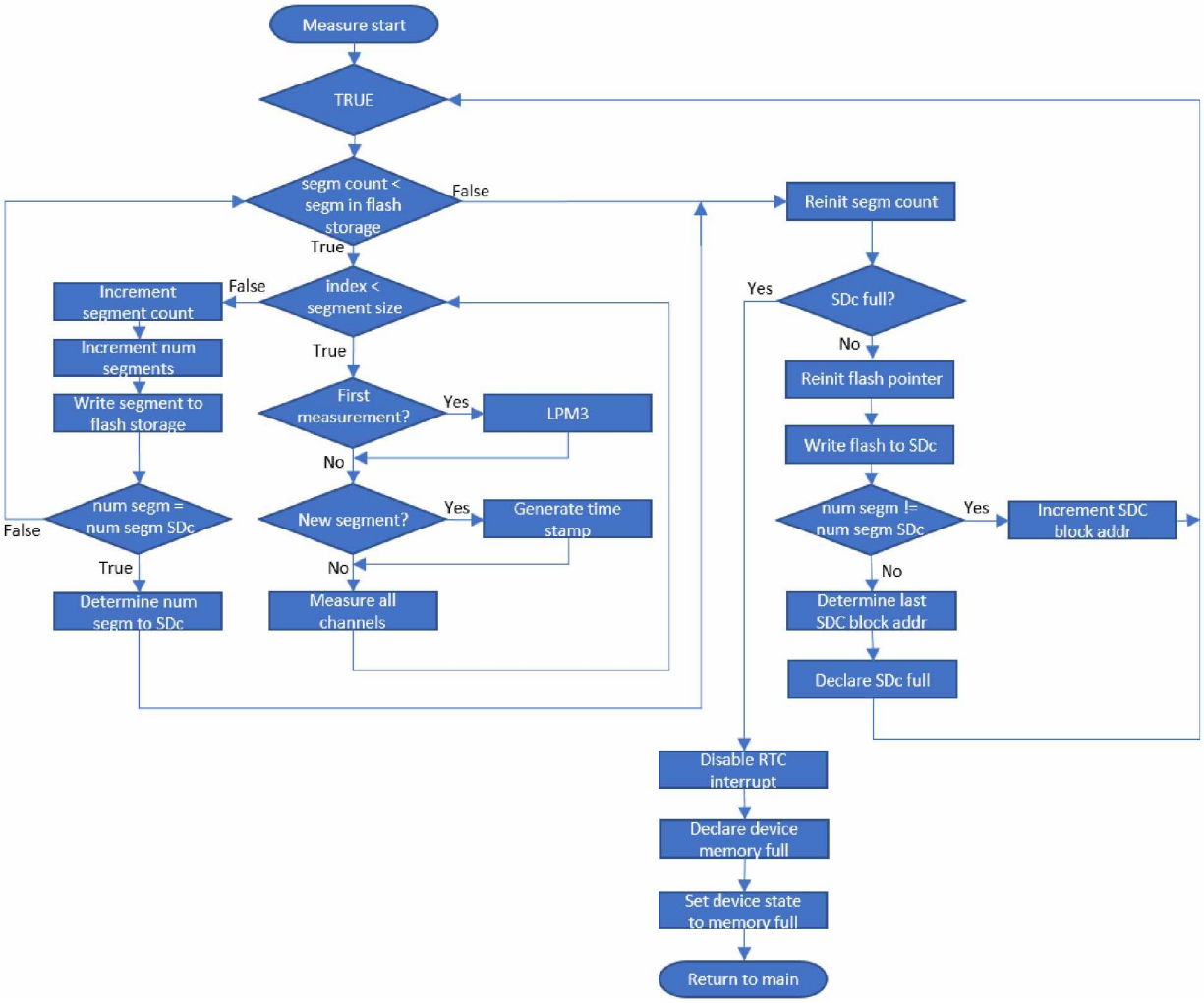

Figure 3-4. Measuring in the "RTC mode" flowchart

3.1.2.2 Measuring in the "Bulk Mode"

The measure_bulk_all_channels_store_flash() function performs the measurement in the "Bulk mode." This function is under development at the moment. Its current implementation is described below.

Each channel is measured continuously until the "num_bulk" number of segments is reached. The "num_bulk" value is set by the user. Data are stored in 512-byte segments each consisting of 170 conversions, or 510 bytes, and 2 bytes of CRC. The measurement data are first written into RAM storage. When RAM storage is full, the segment is written to flash storage. The process repeats until flash storage is full. Writing to the microSD-card is not supported yet.

As mentioned earlier, this function is under development. It lacks several features that will improve its usability. The "Bulk mode" is not fully supported by the user's interface function that

processes the bulk data, so the data should only consist of complete segments. Because of this, the USB hot-plugging functionality is not supported. Currently, the user needs to use a stopwatch to determine the moment when it is safe to connect the Logger to the host PC. The discussed issue will be addressed in the future versions of the firmware and user's software.

3.1.3 The USB Functions

After the Logger has been connected to the host PC, the program finishes the current measurement or wakes up from the LPM3 and calls the usb_connect2host() function, which starts the enumeration process. If the enumeration has finished succesfully, the program goes to the enumerated state (state #1). In this state, the microcontroller enters the LPM0 and waits for the arrival of a command code from the host PC. The command code is the number, which is used to identify the specific communication command.

The USB communication is governed by the host PC. It initiates the communication by sending a command code to the Logger. When the Logger receives the code, an interrupt is generated. The interrupt wakes up the microcontroller, and it proceeds to the usb_command_line() function. This function reads the command code from the USB buffer and decides, which function to call. All USB functions are presented in Table 3-1.

Table 3-1. The USB functions

Function	Description
usb_connect2host()	Connects to the host when the USB VBUS becomes available
usb_command_line()	Calls the appropriate function based on the received command code
usb_ack()	Acknowledges the success of the "set" command
usb_send_conv_result()	Sends measurement result to the host PC
usb_get_dev_param()	Sends a parameter to the host PC
usb_get_meas_rate()	Sends a measurement rate parameter to the host PC
usb_get_adc_cal_register()	Sends the ADC's calibration register to the host PC
usb_get_rtc()	Sends the microcontroller's real-time clock to the host PC
usb_set_dev_param()	Sets a parameter received from the host PC
usb_set_adc_vrefcon()	Sets a VREFCON parameter received from the host PC
usb_set_rtc()	Sets a real-time clock received from the host PC

The usb_send_conv_result() function sends the measurement data to the host PC. The process is divided into two stages. First, the microcontroller sends the host PC the number of bytes of the measurement data, which is stored in the Logger's memory ("num bytes"), then flags this event, and returns to the main function. Second, when the microcontroller receives the same command code, it enters the usb_send_conv_result() function again, but at this time it proceeds directly to sending the

data. The microcontroller sends the data in the following sequence: first, the program moves the data from RAM storage into a temporary array in RAM called “temporary RAM storage,” second, the data from the microSD-card is sent to the PC, third, the data from flash storage is sent to the PC, and finally, the data from the temporary RAM storage is sent to the PC. This order corresponds to the order in which the measurements have been taken. If any of the storage locations does not have the data, the program skips sending the data from there. The usb_send_conv_result() function’s flowchart is shown in Figure 3-5.

The USB “get” and “set” functions are very similar. A “get” / ”set” function is called when the microcontroller receives its code. For a “get” function, the parameter is sent to the host PC. For a “set” function, the value received in the message is assigned to the parameter. The microcontroller acknowledges both “get” and “set” functions.

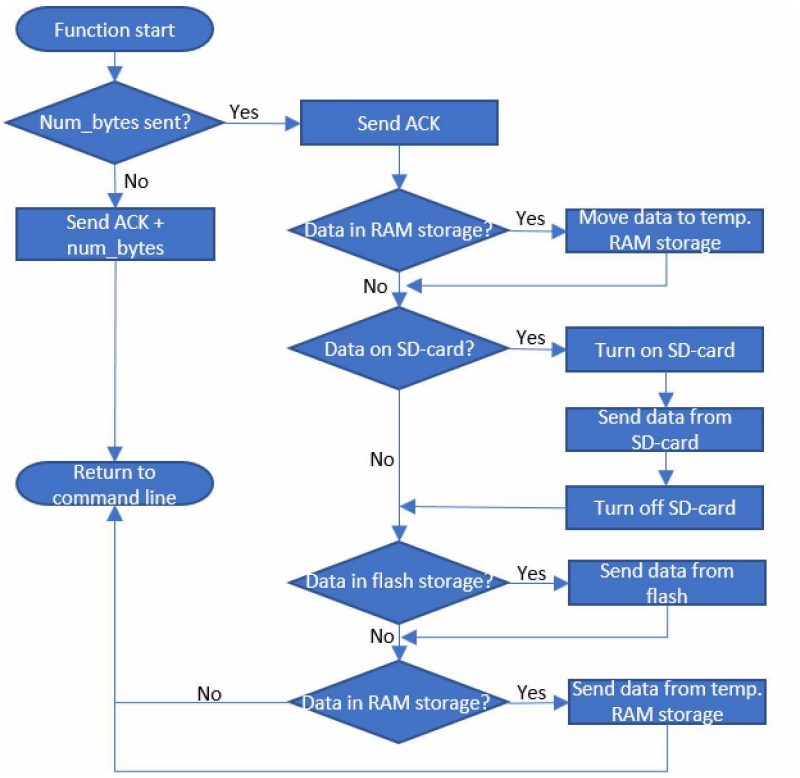

Figure 3-5. Send conversion result function

The “get” / “set” functions for the ADC are slightly different in the way that they need to communicate with the ADC to obtain or set the parameter. The “get” / “set” functions for the real-time clock timer differ from the usual “get” / “set” functions as well. For the “set” function, the microcontroller stops the timer, assigns new values to its registers, and then restarts the timer. For the

"get" function, the microcontroller enables the RTC ready interrupt, which indicates the time when it is safe to read the timer registers, and enters a low power mode. When the timer generates the interrupt, which can occur with a delay of up to 1 second, the microcontroller wakes up, reads the time, and sends it to the host PC.

3.2 The Peripherals' Drivers

3.2.1 Driver for the ADG706 Multiplexer

The multiplexer driver contains functions to set up pins of the MSP430 that control the multiplexer to switch individual channels, to disable the multiplexer, and to choose the input to be connected to the output. The latter function allows stepping through all channels when used in a loop statement.

3.2.2 Driver for the ADS1247 ADC

The analog to digital converter ADS1247 is controlled over an SPI interface [27]. The SPI commands are used for system control, data read, registers read/write and calibration. The list of commands is presented in Figure 3-6, which is reproduced from [27].

COMMAND TYPE	COMMAND	DESCRIPTION	1st COMMAND BYTE	2nd COMMAND BYTE
System Control	WAKEUP	Exit sleep mode	0000 000x (00h, 01h)	
	SLEEP	Enter sleep mode	0000 001x (02h, 03h)	
	SYNC	Synchronize the A/D conversion	0000 010x (04h, 05h)	0000-010x (04,05h)
	RESET	Reset to power-up values	0000 011x (06h, 07h)	
	NOP	No operation	1111 1111 (FFh)	
Data Read	RDATA	Read data once	0001 001x (12h, 13h)	
	RDATAC	Read data continuously	0001 010x (14h, 15h)	
	SDATAC	Stop reading data continuously	0001 011x (16h, 17h)	
Read Register	RREG	Read from register *rrrr*	0010 rrrr (2xh)	0000_nnnn
Write Register	WREG	Write to register *rrrr*	0100 rrrr (4xh)	0000_nnnn
Calibration	SYSOCAL	System offset calibration	0110 0000 (60h)	
	SYSGCAL	System gain calibration	0110 0001 (61h)	
	SELFOCAL	Self offset calibration	0110 0010 (62h)	
Restricted		Restricted command. Should never be sent to device.	1111 0001 (F1h)	

Figure 3-6. ADC SPI commands

Not all commands from the list were implemented in the driver. The utilized START pin (the microcontroller's port 9.0) is the hardware replacement for both SLEEP and WAKEUP commands: setting the START pin "Low" puts ADC into shutdown mode, while setting it "High" starts the conversion. The Logger has only one ADC, therefore the SYNC command, which allows synchronizing the operation of several ADCs, is irrelevant. Additionally, continuous data read with RDATAC/SDATAC commands were

not implemented. This functionality would be useful for a large number of conversions on a single channel, however, since the intended application requires only one conversion per channel at a time, the continuous read function was deemed unnecessary. The ADC driver functions are presented in Table 3-2.

The first four registers of the ADC are MUX0, VBIAS, MUX1, and SYS0. The MUX0 register controls the selection of the positive and the negative inputs of the ADC. The VBIAS register allows applying a bias voltage to thermocouples connected to the ADC's analog inputs; this functionality is not used in the Logger. The SYS0 register controls the PGA and the data rate settings. The MUX1 controls the following modules:

1. The reference voltage generator through the VREFCON bits.
2. The reference input of the ADC through the REFSELT bits.
3. The input multiplexer when it is used for calibration through the MUXCAL bits.

Table 3-2. The ADC driver's functions

Function name	Description
adc_spi_pins_init()	Initializes the START, *RESET, and *Chip Select pins
adc_spi_init()	Initializes the UCA2 module to work in the SPI mode
adc_reset()	Resets the ADC
adc_spi_rreg_single()	Reads a single register
adc_spi_rreg_first4()	Reads the first four registers
adc_spi_rreg_cal_register()	Reads a single calibration register
adc_spi_rreg_all()	Reads all registers
adc_spi_define_wreg_mux0()	Defines the contents of the MUX0 register
adc_spi_define_wreg_mux1()	Defines the contents of the MUX1 register
adc_spi_define_wreg_sys0()	Defines the contents of the SYS0 register
adc_spi_wreg_single()	Writes a single register
adc_spi_wreg_mux0()	Prepares the contents and writes it to the MUX0 register. Uses adc_spi_define_wreg_mux0() and adc_spi_wreg_single()
adc_spi_wreg_mux1_sys0()	Prepares the contents and writes it to the MUX1 and SYS0 registers.
adc_spi_wreg_first4()	Prepares the contents and writes it to the first four registers
adc_setup()	Sets the ADC parameters
adc_spi_rdata_once()	Reads a single conversion
adc_measure()	Performs a conversion for the specified number of times and stores it to RAM.
adc_measure_all_channels()	Performs a conversion of 16 channels, calculates the CRC sum of 16 measurements, stores data to RAM
adc_sysgcal()	Performes the sytem gain calibration
adc_selfocal()	Performes the self offset calibration
adc_sysocal()	Performes the system offset calibration
adc_calibration()	Performes all three types of calibration

The VREFCON bits are used to set the operational mode of the reference voltage generator: "Always OFF", "Always ON", or "ON during conversion." The REFSELT bits allow selecting between the internal and external reference voltages for the ADC's reference input. MUXCAL bits allow performing different measurements: offset and gain calibration, positive and negative analog supply measurements, temperature diode measurements, and measurements on the external reference inputs.

3.2.3 SD-card

The driver for the microSD-card was developed because other drivers available at the time did not work with the MSP430F5659 microcontroller. I used definitions of commands, responses and error codes, as well as some functions' names from the driver created by Texas Instruments [48], but the functions itself were developed from scratch. One useful addition to the developed driver is a multiple block write functionality, which is crucial for the Logger. In case of the single block write, each block of data is accompanied by the overhead. Conversely, for the multiple block write, only the first block of data is sent with the overhead. When all blocks of data are sent, the stop token finishes the transmission. The Logger writes 768 blocks of data at once. For the 2 GB microSD card, it repeats this process 5462 times. Using the multiple block write function saves 767*5462 =4189354 overhead transmissions, which in turn increases the battery life. The microSD-card functions are presented in Table 3-3.

Table 3-3. The microSD-card functions

Function	Description
num_segments_sdc()	Determines the number of segments in the microSD-card. Currently depends on the manually entered value from the microSD-card's datasheet
sdc_CS_Init()	Initializes the *Chip Select pin
sdc_SPI_Init()	Initializes the UCB1 module to operate in the SPI mode
sdcDeactivatePorts()	Configures ports initialized for the USCI function to a general-purpose mode
sdcPowerOn()	Turns on the microSD-card
sdcPowerOff()	Turns off the microSD-card
sdcSendCmd()	Sends a command over the SPI interface
sdcGet_R1_R2_Response()	Receives R1 or R2 responses from the microSD-card
sdcStart()	Initializes the microSD-card in the SPI mode
sdcSetBlockLength()	Sets the block length for a data transfer. The block size is 512 bytes
sdcCheckBusy()	Checks if the microSD-card is busy
sdcGetDataResponse()	Receives the data response token
sdcStopTransmission()	Sends the stop transmission token
sdcReadSingleBlock()	Reads a single block of data
sdcWriteSingleBlock()	Writes a single block of data
sdcWriteMultipleBlocks()	Writes multiple blocks of data
sdcReadRegister()	Reads a single register

Chapter 4 User Interface

4.1 General Information on the User Interface

The user interface is designed as a command line interface (CLI). It operates from within MATLAB workspace. The interface is called by starting the script "cli.m," which performs the following actions. The program requests the user to choose one of the COM ports from the list of available ports; the program will only work if the Logger is already connected to the host PC. After the user has entered the COM-port name, the program tries to establish the communication with the Logger. Once this step is accomplished, the program proceeds to the main loop, which prompts the user to enter the command to be executed. The list of all commands is visible to the user upon entering "help." The commands are presented in Table 4-1. When the user enters a command, the main loop calls the linked function. Once the function's execution is finished, the program returns to the main loop and prompts the user to enter a new command. The main loop execution terminates when the user enters the "exit" command. If the user enters a non-existing command, the program will ask the user to enter the command one more time. The "cli.m" script and all user interface code is available in Appendix G.

Most of the CLI functions require a communication with the microcontroller. This communication is asynchronous or, in other words, an interrupt-driven. Prior to sending any command to the microcontroller, the sending function sets the number of bits that should be received from the Logger in a response to this command. Once the command is sent to the MSP430, the program returns to the main loop. When a response from the microcontroller is received, the interrupt initiates the call of the appropriate callback function.

The communication protocol is simple, yet it has some degree of robustness that is attributed to the acknowledgements system. Every time the host sends a message to the microcontroller, it embeds a code of this command into this message; the command codes are presented in Table 4-1. The microcontroller uses this code to call the appropriate response function. In the return message, the microcontroller sends back the same code. Therefore, the host knows if the communication with the microcontroller has been successful. If the communication has failed (the host has received different code than expected), the program notifies the user about the situation and returns to the main loop, from where the new command can be issued. However, this situation has never happened during the testing.

Table 4-1. The user interface command list

Command	Code	Called function	Description
	"Get" commands		
'get conv result'	127	request_num_bytes()	Requests a conversion result
'get data rate'	3	get_adc_param()	Gets a data rate value
'get meas rate'	25	get_measurement_rate()	Gets a measurement rate value
'get num bulk'	7	get_adc_param()	Gets the number of segments for the "bulk mode"
'get adc vrefcon'	10	get_adc_param()	Gets the operational mode of the ADC's reference voltage generator
'get sysocal en'	19	get_adc_param()	Requests whether the system offset calibration in enabled or disabled
'get delay en'	21	get_adc_param()	Requests whether the delayed start is enabled or disabled
'get keep data'	27	get_adc_param()	Requests whether the data is discarded or kept at the microcontroller
'get cal en'	29	get_adc_param()	Requests if the "calibration always on" feature is enabled or disabled
'get op mode'	30	get_adc_param()	Gets the operational mode information
'get ofc reg'	12	get_adc_cal_register()	Requests the ADC's OFC register
'get fsc reg'	13	get_adc_cal_register()	Requests the ADC's FSC register
'get rtc'	17	get_rt_clock()	Requests the real time clock value from the microcontroller
"Set" commands			
'set meas rate'	23	set_measurement_rate()	Sets a measurement rate value
'set meas start'	24	set_measure_start()	Sets a measurement start option
'set data rate'	5	set_adc_data_rate()	Sets a data rate value
'set num bulk'	9	set_num_of_bulk_segments()	Sets the number of segments for the "bulk mode"
'set adc vrefcon'	11	set_adc_vrefcon_reg()	Sets the operational mode of the reference voltage generator
'set sysocal en'	20	set_sysocal_enable()	Set the operational mode of the ADC's reference voltage generator
'set delay en'	22	set_delay_enable()	Enables the "delayed start" feature
'set rtc'	18	set_rt_clock()	Sets the real-time clock on the microcontroller
'set keep data'	26	set_keep_data_in_memory()	Sets the "keep data" feature to "On" or "Off"
'set cal en'	28	set_cal_enable()	Sets the "calibration always on" feature to "On" or "Off"
'set op mode'	31	Set_operational_mode()	Sets the operational mode to either "rtc_mode" or "bulk_mode"
"Help" commands			
'help'	-	cli_help()	Provides the list of all commands
'help vrefcon'	-	cli_help_vrefcon()	Provides help on the "vrefcon" cmd
"Escape" commands			
'abort'	-	fclose()	Aborts the last sent command
'exit'	-	Break the main loop	Exits from the CLI

The communication functions were designed in such a way, that every action pertaining to the communication is explicitly acknowledged, so it is easy to control each stage of the data transfer process.

Ultimately, the command line interface will be replaced with a graphical user interface (GUI). This transition should be easy, because only the main program will change, while the underlying architecture will stay the same.

4.2 Get Conversion Result

To get a measurement result from the Logger, the user enters the 'get conv result' command. This command calls the function request_num_bytes(), which sends a request for the number of bytes of the measurement data that the microcontroller will send to the MATLAB script, and then returns to the main loop. Once the number of bytes of the measurement data is received, the num_bytes_RXed_callback() function is called. Based on the number of bytes, the function allocates the host PC's buffer size, sends the request for the data to the microcontroller, and returns to the main loop.

When all data are received into the buffer, the data_received_callback() is called. This function moves the data into workspace and calls either the ProcessInputData() or the ProcessInputBulkData() function. The former processes the measurement data obtained in the "RTC Mode", the latter processes the data obtained in the "Bulk Mode," as discussed in 3.1.2.1 and 3.1.2.2 respectively. Two separate functions to process the data are required because the data format is not the same for the Logger's operational modes.

As mentioned in 3.1.2.1, time stamps are generated for a segment and not for each individual measurement of 16-channels. The ProcessInputData() function generates the time stamp for each measurement. First, it finds the first valid time stamp by checking its CRC value. If the valid time stamp is the first in the data stream , the function adds the measurement interval to the time stamp until it reaches the data end. If the valid time stamp is not the first in the data stream, the function subtracts the measurement rate from the time stamp until it reaches the beginning of data, and then it adds the measurement rate starting form the initial time stamp until it reaches the data end.

Once the data processing is finished, regardless of the function that has been used to accomplish it, the PresentConvResult() function is called to calculate the measured resistance, temperature, and voltage from the binary-coded data, present it to the user, and save the data to the host PC. The measured resistance and voltage are determined using equations 2.4 and 2.3 respectively. The measured temperature is calculated using interpolation as discussed below.

4.2.1 Converting the Measured Resistance into Temperature

Usually, a thermistor resistance is converted to temperature using the Steinhart-Hart (S-H) equation [49] presented below:

where T is the temperature in Kelvin, R_{TH} is a thermistor resistance, A, B, and C are the coefficients of the equation (constants).

For the PS103J2, the manufacturer provides the coefficients that optimize the S-H equation in the range from 0° C to 76° C. The goal of optimization is to provide a minimal conversion error in the range of interest. The conversion error can be calculated as follows: first, the temperature is determined by substituting resistances from the thermistor's RT-data into the S-H equation and, second, the obtained temperatures are compared to temperatures in the thermistor's RT-data. In the range 0° C to 76° C, the conversion error varies from -0.1 to 0.2 mK (Figure 4-1.(right)). For temperatures below 0° C, the S-H equation yields the conversion error up to 96° mK at -40° C (Figure 4-1.(left)). Therefore, it was necessary to find more appropriate coefficients for the temperature range of interest -40° C to 40° C. This was done by solving the system of three equations 4.1 for different operating points of the thermistor. Operating points including but not limited to (1) -40°C, 0°C, 40°C and (2) -20°C, 0°C, 20°C were used. Calculations were performed in MATLAB.

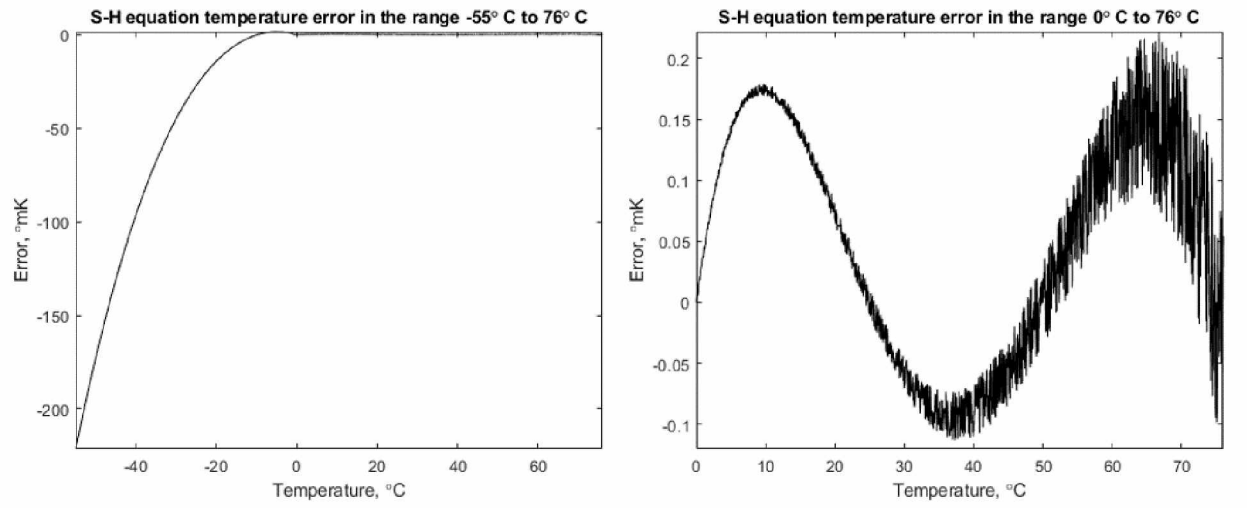

Figure 4-1. The S-H equation error in the range of -55° C to 76° C (left) and 0° C to 76° C (right)

As can be seen from Figure 4-2.(left), for the operating points -40°C, 0°C, 40°C, the conversion error approaches zero if it is closer to the operating point temperatures and departs from zero if it is further from the operating point temperatures. The maximum error is 0.0082° C, which is almost the

same as the required accuracy. The conversion error for the operating points -20°C, 0°C, 20°C is shown in Figure 4-2.(right).

In Figure 4-2, there is a steep drop in the error values at temperatures from -1.05° C to 1.00° C. The same drop is present in Figure 4-1.(left). This drop is caused by the behavior of the PS103J2 RT-characteristic at these temperatures, as can be seen in Figure 4-3, which shows the difference between the adjacent resistance values in the range -1.15° C to -0.95° C.

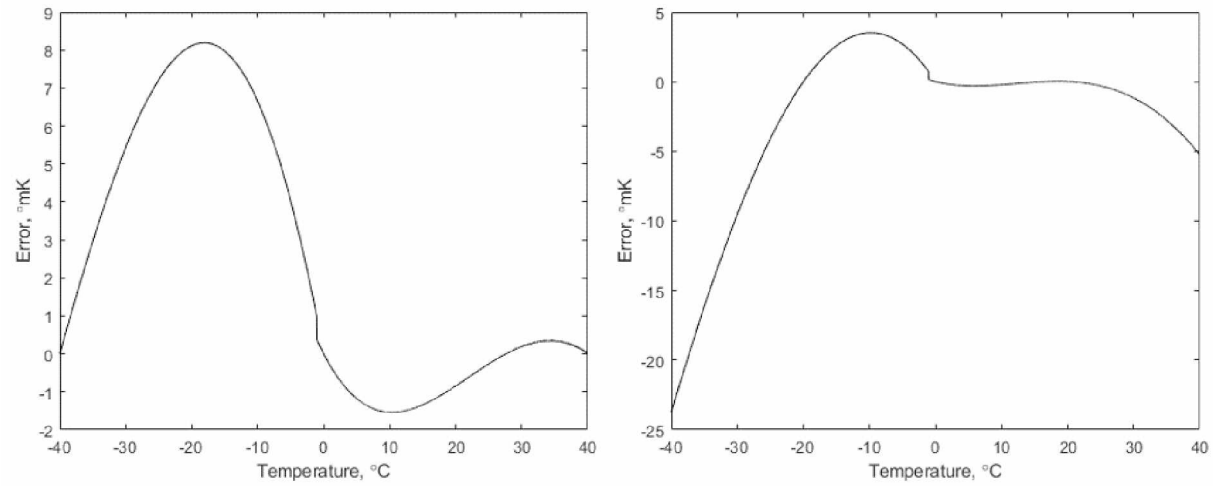

Figure 4-2. The S-H equation error for operating points -40°C, 0°C, 40°C (left) and -20°C, 0°C, 20°C (right)

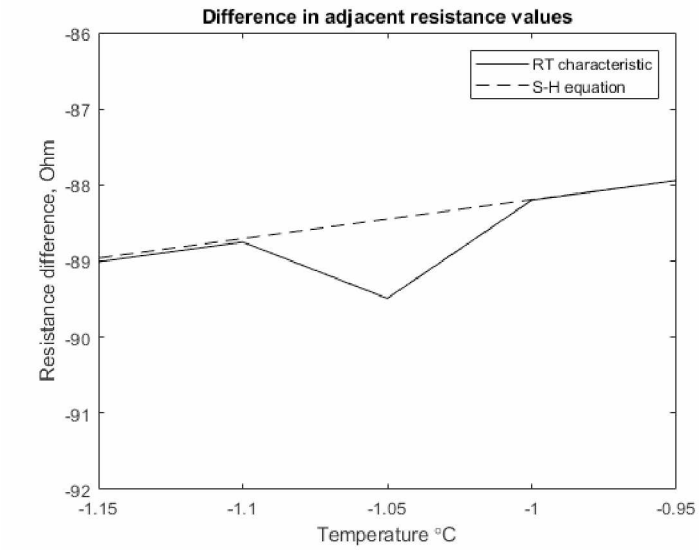

Figure 4-3. The difference in adjacent resistances in the range -1.15° C to -0.95° C for the PS103J2 thermistor

Selecting different operating points, I was not able to find the S-H equation coefficients that would produce an acceptable error in the temperature range of interest. For this reason, I decided to use interpolation instead of the S-H equation. Measured resistances are converted to temperature using piecewise cubic Hermite interpolation (PCHIP) [50]. The interpolating function passes through all the points of the thermistor's R-T curve, thus maintaining a zero error at these points.

4.3 Functions to Get the Logger Parameters

Several different functions are used to obtain the Logger's parameters and other information. The get_adc_param() function is used to request the value of many parameters (Table 4-1). The get_adc_cal_register() function instructs the Logger to send the ADC' calibration register, either OFC or FSC. The get_rt_clock() function allows to obtain the Logger's real-time clock.

All "get" functions share the same architecture. After the function is called, it prompts the user to enter the value of the parameter. Then it prepares the host PC's buffer for the microcontroller's response, sends the message that consists of the command code and the parameter's value, and returns to the main loop. The appropriate callback function is called upon receiption of the microcontroller's response. The callback function checks the returned command code: if the code is valid, it displays the parameter's value to the user; if the code is not valid, the function notifies the user and returns to the main loop.

4.4 Functions to Set the Logger Parameters

"Set" functions are used to configure different parameters of the Logger and the ADC. Brief descriptions of all "set" commands are presented in Table 4-1. Although "set" functionality is implemented with different functions, all these functions share the same architecture, as described below. First, the function prompts the user to enter the value of the parameter to be set. When doing this, it checks the validity of entered data, and asks the user to retry if the entered value is invalid. All functions give a clue on the valid value. Second, once the value is accepted, the message to be sent to the microcontroller is generated. This message contains the code of the command and the parameter value itself. Finally, the function passes the message to the send_set_cmd() function, which will send it to the microcontroller.

All set commands use the same function send_set_cmd() to send messages to the microcontroller. After the message has been sent, this function sets the number of bytes for the microcontroller's acknowledgement and returns to the main loop. When the microcontroller receives the message, it acknowledges it by sending the command's code back to the program. The

acknowledgement is processed in a callback function, which notifies the user whether the command went successfully and returns to the main loop. The user has an option of issuing the “get” command to verify that the parameter has been set successfully.

Chapter 5 Experimental Results

5.1 General Information on Experiments

Several different experiments were conducted to evaluate the performance of the Logger and compare it with the performance of the Campbell Scientific CR1000 – a logger that is widely used by researchers at the UAF Geophysical Institute and other research universities in the US and worldwide. The experiments included determining the equivalent temperature accuracy with a known resistive input, studying the effect of ambient temperature on accuracy of resistance measurements (for the Logger only), performing an ice-bath measurement (for the Logger only), and measuring air temperature using 10 kΩ thermistors.

Ideally, the temperature accuracy of the Logger and the CR1000 should have been determined by comparison with a calibrated thermometer. However, since no such thermometer was available, an equivalent temperature accuracy was determined instead. The method is based on measuring some known resistance with the device under test and a reference ohmmeter, converting the measured values into temperatures and comparing them. In this case, the measured accuracy incorporates the error of the reference ohmmeter.

To represent the range of -40° to 40° C, 17 thin film surface mount (SMT) and 1 bulk metal through hole (TH) resistors with low temperature coefficients were used. Resistor specifications are presented in Table 5-1. The resistors were mounted on a custom board that allowed individual or multiplexed measurements (in this case the board emulated a thermistor probe). The board is shown in Figure 5-1; the design details can be found in Appendix C. First, resistances were measured with the Keysight 34420A Nano Volt / Micro Ohm Meter, the Keysight 34410A/11A 6½ Digit Multimeter and the Fluke 8845A Digital Multimeter utilizing the 4-wire method. Since results were almost identical for all multimeters, only the Keysight 34420A was used as a reference ohmmeter subsequently. Second, resistances were measured with the Logger and the CR1000. Finally, resistances measured by loggers and the Keysight 34420A were converted to temperature using interpolation onto resistance-temperature curves of 10 and 20 kΩ thermistors, and the results were compared to provide the equivalent temperature accuracy estimations.

In the experiment described above, equipment was at room temperature (24° C). To study the effect of ambient temperature on measurement accuracy of the Logger, the experiment was repeated at -30°, -25°, -20° C in the chest freezer and 6° C in the refrigerator. In all tests, the ambient temperature was controlled and logged using the Hobo UX120-006M logger with a built-in LCD.

Table 5-1. Resistors for the accuracy measurements

Resistor Model	Type	Nominal Resistance, kΩ	Tolerance, %	Temperature Coefficient, ppm/°C	Represented temperature for 20K thermistor, °C	Represented temperature for 10K thermistor, °C
ERA-6AEB6653V	SMT	665	±0.1	±25	-39.85	Outside the range
ERA-6AEB4873V	SMT	487	±0.1	±25	-35.05	-45.45
RN73C1J357KBTD	SMT	357	±0.1	±10	-30.15	-40.90
RN73C1J261KBTD	SMT	261	±0.1	±10	-25.05	-36.15
RN73C1J196KBTD	SMT	196	±0.1	±10	-20.15	-31.65
RN73C1J147KBTD	SMT	147	±0.1	±10	-15.15	-26.95
RN73C1J110KBTD	SMT	110	±0.1	±10	-9.90	-22.15
RN73C2A84K5BTDF	SMT	84.5	±0.1	±10	-4.95	-17.60
Y145360K0000V9L	TH	60	±0.005	±0.2	1.65	-11.50
ERA-3ARB513V	SMT	51	±0.1	±10	4.90	-8.50
ERA-3ARW393V	SMT	39	±0.05	±10	10.40	-3.45
RN73C2A31K6BTDF	SMT	31.6	±0.1	±10	14.90	0.65
RN73C2A24K9BTDF	SMT	24.9	±0.1	±10	20.05	5.40
RN73C2A20KBTD	SMT	20	±0.1	±10	25.00	9.90
ERA-3ARB163V	SMT	16	±0.1	±10	30.15	14.60
ERA-3ARB133V	SMT	13	±0.1	±10	35.10	19.10
RN73C2A10K7BTDF	SMT	10.7	±0.1	±10	39.90	23.45
RN73C2A4K99BTD	SMT	4.99	±0.1	±10	Outside the range	41.60

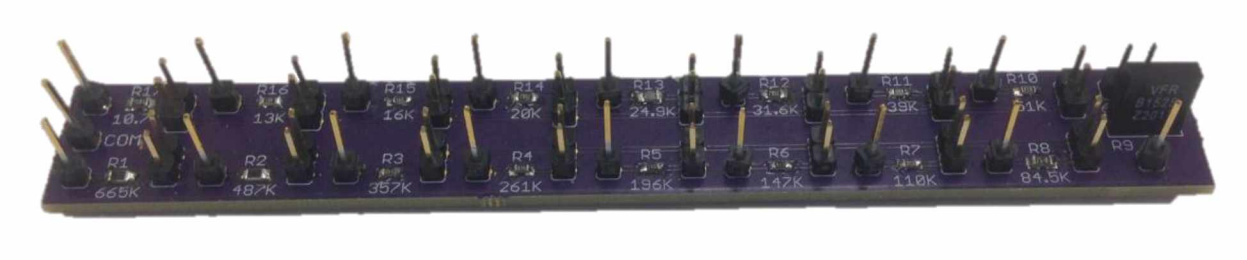

Figure 5-1. Resistive test bed

Data from the resistance measurements at room temperature were used to calibrate the Logger. To verify the calibration accuracy, four more resistors were measured at room temperature: one bulk metal TH resistor (475 kΩ ±0.1% ±2.5 ppm/°C), and three thick film SMT resistors (124 kΩ, 47 kΩ, 33 kΩ, all ±0.1% ±100 ppm/°C).

To compare the Logger and the CR1000 performances in real-life scenarios, several experiments to measure air temperature indoors, outdoors, and in the freezer were conducted. U.S. Sensor Corp. PS103J2 10 kΩ 0.1° C interchangeable thermistors were used as sensing elements. The combination of the Logger and the thermistors was not calibrated; therefore, no accuracy estimations could be provided from these experiments.

A "half-bridge" scheme, as described in the CR1000's user guide, was used for measurements with the CR1000. This scheme is identical to configuration #1 discussed in 2.1.2. Researchers in GIPL use a Vishay USR2-0808 25 kΩ 1 ppm/°C reference resistor, which is why it was chosen for the experiments. In half-bridge setup, voltage across a resistor is measured with respect to ground potential, therefore the CR1000 specifications pertaining to single-ended measurements were used to assess the results. Even though the CR1000 has 16 channels, the interconnections to utilize them when measuring thermistors would have been cumbersome and potentially unreliable. Therefore, an AM416 Relay Multiplexer was used for thermistors connections. The test bed with the CR1000, the Relay Multiplexer and the battery is shown in Figure 5-2.

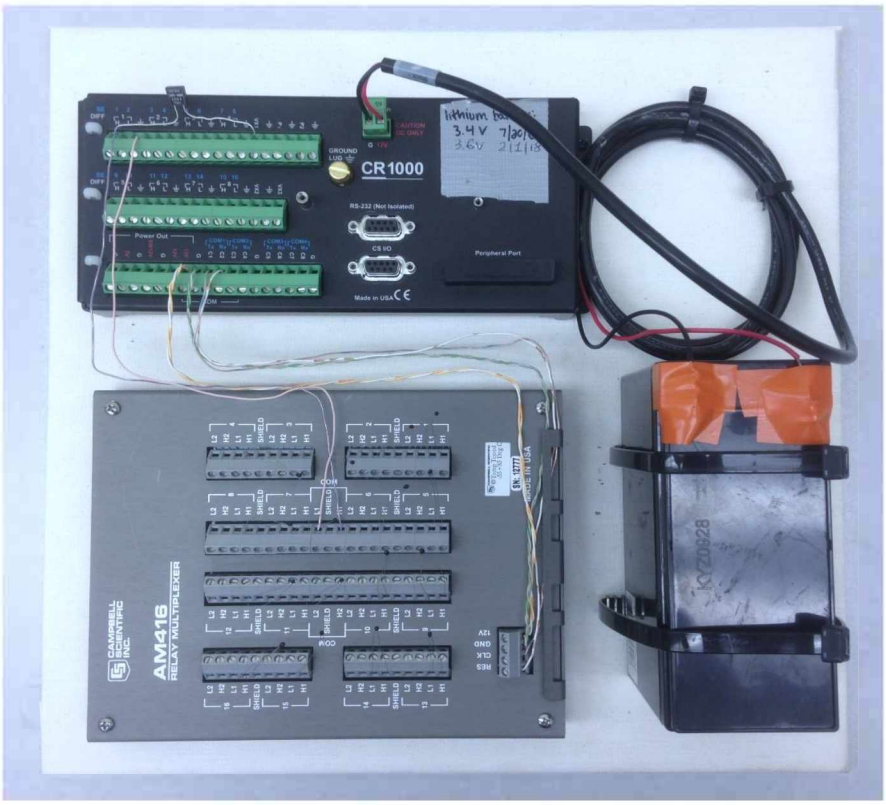

Figure 5-2. The CR1000 test bed

In the early experiments, I noted that several factors were contributing to the measurements' noise: (1) connection to the PC through the communication interface (e.g., USB for the Logger), (2) length and configuration of cables connecting resistors to the loggers, (3) proximity to electric devices (a PC, a monitor, etc.), (4) motions of the experimenter.

Therefore, efforts were made to remove these factors. First, measurements were made with communication interfaces disconnected. Second, short twisted cables were used to connect resistors to loggers. Third, loggers were placed further away from electrical devices for the duration of measurement sessions. Finally, to eliminate the fourth factor the following strategies were utilized: taking the large number of samples with the intent of removing outliers in the beginning of the measurement session (for the Logger and the CR1000) or delaying the session start (for the Logger). The latter was implemented in the ad-hoc version of firmware. To further reduce electrical noise, data rates of 5 SPS and 10 SPS were set for the Logger. At these data rates, the ADC's digital filter provides the best attenuation of 60 Hz noise [27]. For the same reason, the CR1000 integration time was set to 16.67 ms [51].

During the resistance measurements, the Logger operated in the "Bulk Mode", as described in 3.1.2.2, providing the conversion result every 200 ms for the data rate of 5 SPS, or 100 ms for the data rate of 10 SPS until the specified number of measurements for each channel had been met; the CR1000 was making measurements every 100 ms for a single channel until the experimenter had stopped it. During the air temperature measurements, both the Logger and the CR1000 were making measurement once every 6 seconds for all channels until the experimenter had stopped the loggers.

5.2 Equivalent Temperature Accuracy with Resistive Input

5.2.1 Measurements at Room Temperature

Equivalent temperature accuracies of the Logger (Board 1 with the single channel, and Board 2 with 16 multiplexed channels) and the CR1000 were determined in the series of experiments at room temperature. Throughout the measurements, ambient temperature was logged with Hobo UX120. Data from Hobo UX120 showed that short-term temperature fluctuations did not exceed 0.5° C. These data were later used to assess the resistors' drift. The following subsections provide detailed information on the experiments. The temperature swing for the entire experiment duration did not exceed 2° C (22.5° C to 24.5° C). Measurements were conducted in series: each resistor was first measured with the reference ohmmeter, then with the Logger, and finally with the CR1000; after this, the sequence was repeated for the next resistor.

5.2.1.1 Resistance Measurements with the Logger

As mentioned earlier, Board 2 is a fully functional prototype with 16 channels, therefore experiments were initially conducted with this board. According to the ADG706 multiplexer datasheet [52], maximum ON resistance match between channels is 0.8 Ω, which should yield negligible difference in channel-to-channel results. Therefore, all measurements were made for channel 1 only.

For Board 2, 8160 samples at data rate of 10 SPS were taken for resistors 10.7 – 109 kΩ. An example distribution of 8160 samples' measurement of the 10.7 kΩ resistor is shown in Figure 5-3. The resistance varies by 0.0097 % around average value. The large number of samples presumably would have provided drift-free intervals for further statistical analysis. However, except for one case (16 kΩ resistor), no significant variations in measured values were observed, and for subsequent measurements (resistor 5 kΩ, 147 – 665 kΩ) the number of samples was reduced to 5100. The data rate of 10 SPS was chosen to make experiments faster assuming it was providing an excellent attenuation of the 60 Hz noise.

The remaining measurements were mostly drift-free except for 16 kΩ and 147 kΩ resistors (Figure 5-4) with the drift of 0.35 Ω and 2.31 Ω respectively; in both cases, the amount of drift matches the room temperature change during the tests. For some measurements the first few samples were beyond the typical range of the consequent samples: in Figure 5-5, for the 24.5 kΩ resistor, sample #30 is 6.2 Ω below the average value for this session, and for the 109 kΩ resistor, sample #3 is 16.75 Ω below the average value for the session. To obtain the statistical data for resistance (Table 5-2), from 5100 samples 5000 consequent samples representing drift- and outlier-free regions were used.

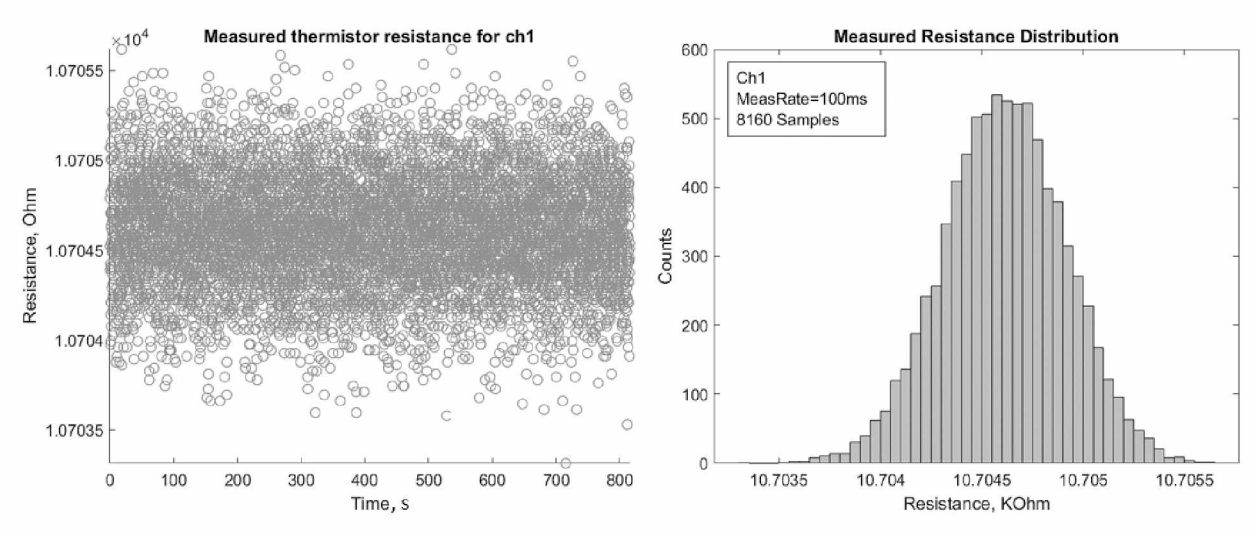

Figure 5-3. 8160 samples for the 10.7 kΩ resistor. Board 2, DR = 10 SPS

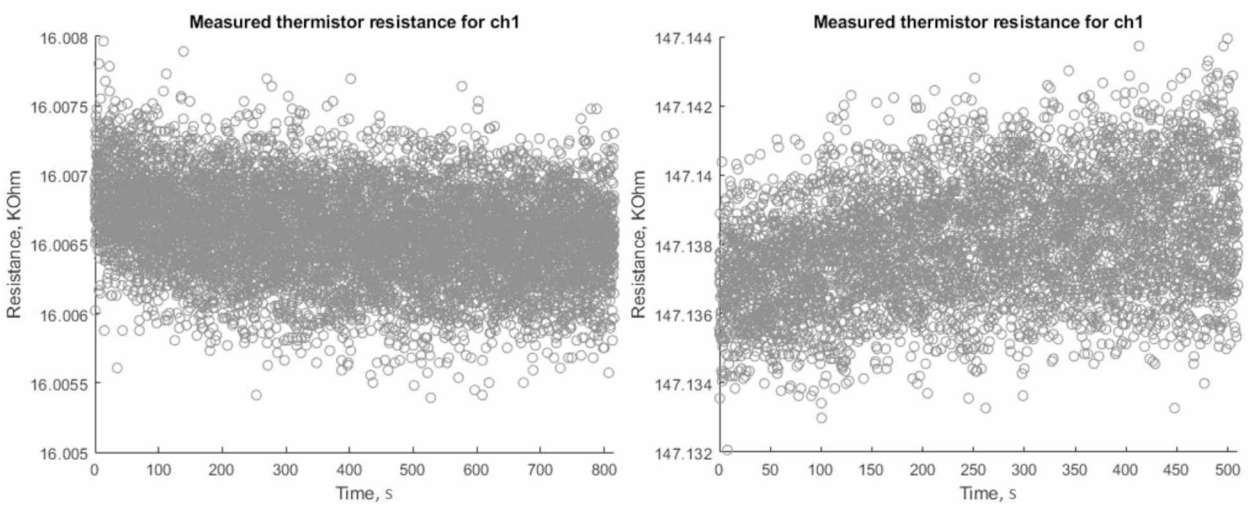

Figure 5-4. Drift of the 16 kΩ (left) and the 147 kΩ (right) resistors. 8160 and 5100 samples respectively. Board 2, DR = 10 SPS

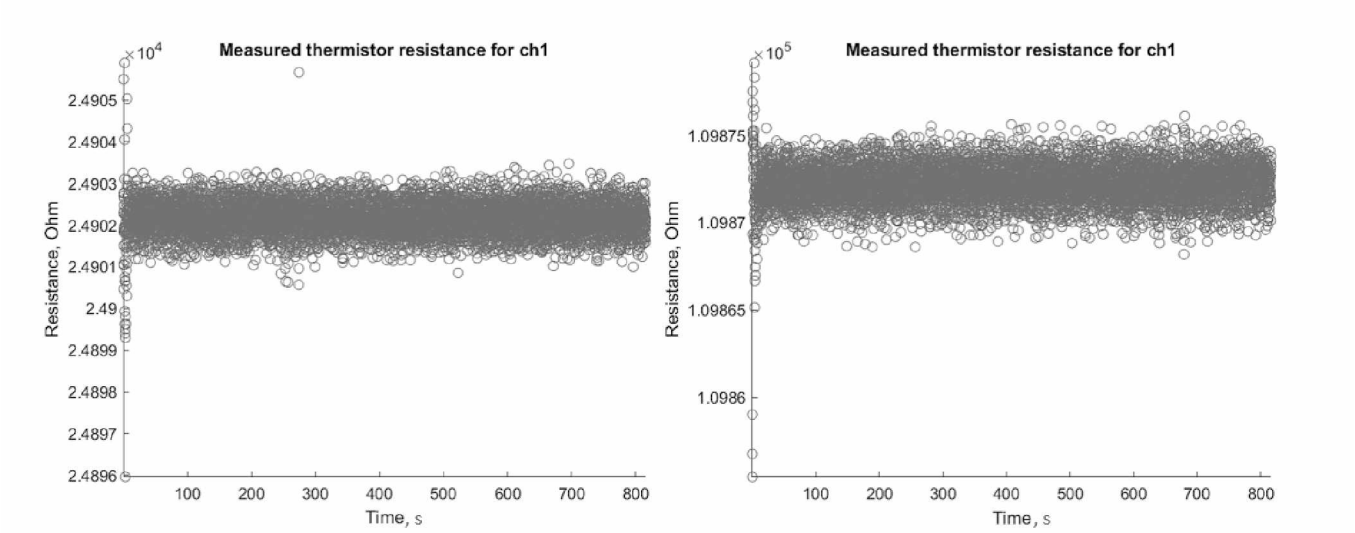

Figure 5-5. Outliers at the session start for the 24.5 kΩ (left) and the 109 kΩ (right) resistors. Board 2, DR = 10 SPS, 8160 samples

As can be seen from the histogram in Figure 5-3, the distribution of the measured resistance resembles normal with values centered around the average. Measured resistances for all resistors were distributed similarly, therefore standard deviation was calculated and included in Table 5-2.

Table 5-2. Statistics for measured resistances for Board 2, DR = 10 SPS, 5000 samples

Resistor ID	Ideal[3] resistance, Ω	Measured resistance, Ω				
		Min deviation	Average	Max deviation	Standard deviation	Difference between average and ideal resistances, Ω / %
665K	665405	-42.2253	666727.45	48.7092	13.0809	1322.45 / 0.1987
487K	487418	-27.2192	488160.39	28.8534	8.0163	742.39 / 0.1523
357K	356756	-24.7061	357190.25	23.9373	7.0181	434.25 / 0.1217
261K	261052	-9.8750	261314.36	11.0021	3.3076	262.36 / 0.1005
196K	196078	-8.3052	196246.91	7.8037	2.5196	168.91 / 0.0861
147K	147032	-5.0959	147138.08	5.8557	1.7165	106.08 / 0.0721
110K	109802	-3.5771	109872.18	3.4500	1.0456	70.18 / 0.0639
85K	84516	-4.2064	84560.68	5.2276	0.8665	44.68 / 0.0529
60K	60001.5	-2.0185	60026.22	2.2564	0.6149	24.72 / 0.0412
51K	50998.9	-2.316	51020.05	1.9078	0.5378	21.15 / 0.0415
39K	39001.5	-1.5898	39014.30	1.4068	0.4771	12.80 / 0.0328
32K	31597.6	-1.5355	31607.82	1.4300	0.4199	10.22 / 0.0323
25K	24894.3	-1.3052	24902.16	1.3162	0.3687	7.86 / 0.0316
20K	19997.8	-1.251	20003.87	1.3448	0.3442	6.07 / 0.0304
16K	16002.0	-1.1361	16006.53	1.1662	0.3280	4.53 / 0.0283
13K	13001.1	-1.1327	13004.49	1.2556	0.2979	3.39 / 0.0261
11K	10701.6	-1.0337	10704.61	1.0054	0.3010	3.01 / 0.0281
5K	4982.33	-0.9913	4983.62	0.9443	0.2724	1.29 / 0.0259

From experiments with Board 2, it was observed that results for 5000 and 510 drift- and outlier-free samples were almost identical. An example data for 10.7 kΩ and 261 kΩ resistors is provided in Table 5-3: for 510 samples, the range of values is slightly smaller, but the average and standard deviation values are almost the same. Based on this fact, for subsequent experiments the number of samples was reduced to 1700, out of which 510 consequent drift- and outlier-free samples were used to obtain the statistical data.

Table 5-3. Comparison of measurements for 5000 and 510 samples

Resistor ID	Difference in average values, Ω	Difference in range, Ω	Difference in standard deviation, Ω	Samples range
261K	0.1327	0.8759	-0.1849	4491:5000
10.7K	-0.0372	0.2984	-0.0052	1:510

The tabulated measured voltage data are presented in Table E-1 in Appendix E. For each resistor, measured voltages varied within ±17 µV from average value (34 µV peak-to-peak). This variation can be defined as a peak-to-peak noise voltage of the measurement. According to the ADS1247

[3] The resistance measured with the reference ohmmeter is called “ideal” in the tables.

datasheet (Table 5 in [27]), the peak-to-peak input-referred noise voltage of the ADC for the data rate of 10 SPS, gain of 1 and a supply voltage of 3.3 is 16.85 μV, which is twice smaller than measured with Board 2. For consequent measurements, I decided to use the data rate of 5 SPS in addition to 10 SPS because at 5 SPS the digital filter has slightly different transfer function than at 10 SPS (as shown in Figures 54 and 55 in [27]), which could have helped to reduce the noise voltage. Unfortunately, due to improper handling of Board 2 in one of the first experiments with the new data rate, it was subjected to an electrostatic discharge. As the result, elements of the power supply circuit were destroyed making further tests with Board 2 impossible. All further experiments were conducted with Board 1.

Board 1 resistance data for 10 SPS and 5 SPS are presented in Table 5-4 and Table 5-5 respectively. Resistance measurement data followed a normal distribution for all resistors.

Table 5-4. Statistics for measured resistances for Board 1, DR = 10 SPS, 510 samples

Resistor ID	Ideal resistance, Ω	Measured resistance, Ω				
		Min deviation	Average	Max deviation	Standard deviation	Difference between average and ideal resistances, Ω / %
665K	665405	-47.0719	666662.31	47.6080	16.5434	1257.31 / 0.1890
487K	487418	-25.5274	488099.53	25.3006	9.7087	681.53 / 0.1398
357K	356756	-15.7486	357137.12	15.6785	5.357	381.12 / 0.1068
261K	261052	-8.1309	261280.00	7.7792	2.4448	228.00 / 0.0873
196K	196078	-6.6214	196223.21	6.4835	2.2973	145.21 / 0.0741
147K	147032	-4.1467	147126.63	4.5282	1.5332	94.63 / 0.0644
110K	109802	-3.5009	109858.96	4.7837	1.3961	56.96 / 0.0519
85K	84516	-2.5239	84552.82	2.0684	0.8437	36.82 / 0.0436
60K	60001.5	-2.7702	60022.84	2.5889	0.9922	21.34 / 0.0356
51K	50998.9	-1.8434	51016.17	1.8054	0.6613	17.27 / 0.0339
39K	39001.5	-1.7367	39013.25	1.2845	0.5163	11.75 / 0.0301
32K	31597.6	-1.2799	31606.75	1.3259	0.4854	9.15 / 0.0290
25K	24894.3	-1.3425	24902.06	1.3401	0.4218	7.76 / 0.0312
20K	19997.8	-0.8846	20004.24	1.0049	0.3563	6.44 / 0.0322
16K	16002.0	-1.4141	16006.80	1.1219	0.3495	4.80 / 0.0300
13K	13001.1	-0.7981	13003.73	0.9715	0.2785	2.63 / 0.0202
11K	10701.6	-0.9174	10705.41	0.9889	0.2786	3.81 / 0.0356
5K	4982.33	-0.6248	4983.11	0.7209	0.2372	0.78 / 0.0157

According to the ADS1247 datasheet (Table 5 in [27]), for the data rate of 5 SPS, gain of 1 and a supply voltage of 3.3 V, the peak-to-peak input-referred noise voltage is 14.24 μV. At 5 SPS average noise voltage for all measurements was 16 μV (the measured voltage data are presented in Table E-3 in Appendix E), which corresponds to the value from the datasheet and is approximately half the noise voltage at the data rate of 10 SPS. As mentioned earlier, this is most likely caused by the difference in transfer functions of the digital filter at 5 SPS and 10 SPS. Due to smaller noise voltage at 5 SPS, the

resistance data had smaller standard deviation than in the case of 10 SPS, and the average resistances were closer to the ideal resistances.

Table 5-5. Statistics for measured resistances for Board 1, DR = 5 SPS, 510 samples

Resistor ID	Ideal resistance, Ω	Measured resistance, Ω				
		Min deviation	Average	Max deviation	Standard deviation	Difference between average and ideal resistances
665K	665405	-16.4379	666614.74	21.1789	6.8565	1209.74 / 0.1818
487K	487418	-12.5661	488081.34	13.5939	4.4870	663.34 / 0.1361
357K	356756	-7.6705	357132.71	7.4692	2.8676	376.71 / 0.1056
261K	261052	-4.4802	261278.95	5.1534	1.8413	226.95 / 0.0869
196K	196078	-4.4641	196220.06	3.4387	1.5353	142.06 / 0.0725
147K	147032	-2.4237	147121.00	2.6245	0.8507	89.00 / 0.0605
110K	109802	-1.6511	109858.89	1.5473	0.5365	56.89 / 0.0518
85K	84516	-1.3159	84550.66	1.4146	0.5448	34.66 / 0.0410
60K	60001.5	-0.9090	60019.51	1.0687	0.3422	18.01 / 0.0300
51K	50998.9	-0.8230	51013.79	0.9583	0.3288	14.89 / 0.0292
39K	39001.5	-0.7668	39011.34	0.7933	0.2817	9.84 / 0.0252
32K	31597.6	-0.7395	31604.86	0.7206	0.2698	7.26 / 0.0230
25K	24894.3	-0.7376	24900.25	0.5321	0.2273	5.95 / 0.0239
20K	19997.8	-0.5493	20002.42	0.5768	0.2144	4.62 / 0.0231
16K	16002.0	-0.6555	16005.16	0.6395	0.2176	3.16 / 0.0197
13K	13001.1	-0.5567	13003.49	0.6459	0.1900	2.39 / 0.0184
11K	10701.6	-0.5413	10703.64	0.5196	0.1899	2.04 / 0.0191
5K	4982.33	-0.4367	4983.21	0.4554	0.1598	0.88 / 0.0177

The difference (in Ω and %) between average resistances measured with Board 2 and Board 1 was calculated to verify the reproducibility of results obtained with two boards (Figure 5-6). Matlab simulations showed that the difference can be attributed to the tolerance of reference resistor and the fact that one of the boards does not have the multiplexor. It is obvious that two ADCs do not match ideally, which also contributes to the difference.

The difference between resistances measured with the Logger (both boards) and the reference ohmmeter increases with the value of resistance. An example for the data rate of 10 SPS is presented in Figure 5-7. This effect is caused by the fact that the ADC's PGA input resistance is not infinitely large and that it depends on the input voltage (this is also noted in Table 7 in [27]). To verify this assumption, a circuit similar to the ADC's PGA circuit was simulated in PSPICE (Appendix D).

As presented in Appendix D, the simulation results were close to the experimental data. If the PGA's input resistance was known this effect could have been eliminated. However, as discussed in

Appendix D, the task of determining the input resistance is nontrivial. Therefore, instead of measuring this resistance, the calibration technique described below was implemented.

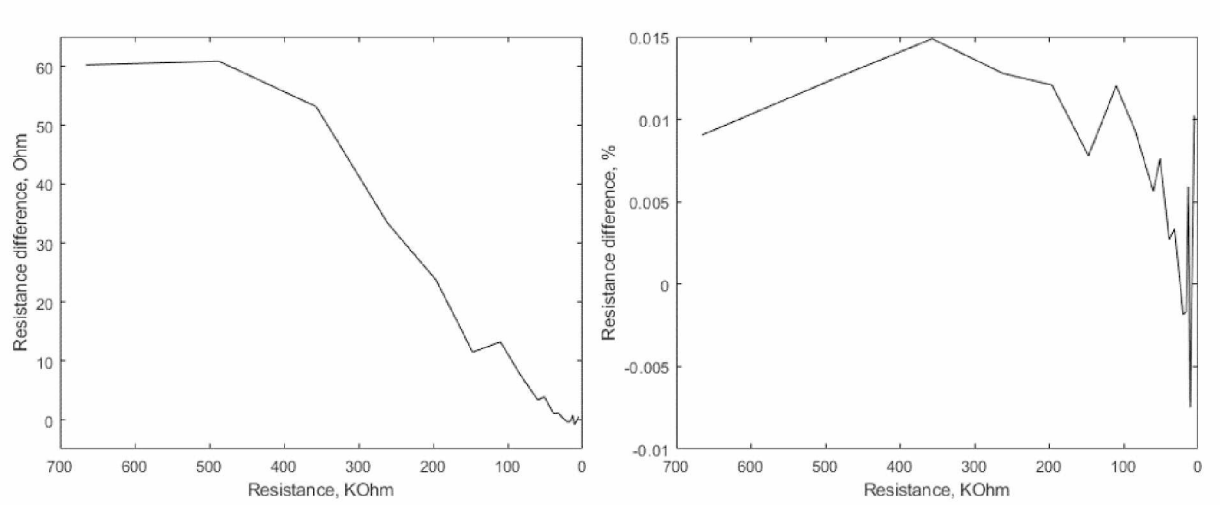

Figure 5-6. Difference in average measured resistance in Ω (left) and % (right) and percent difference between the two boards

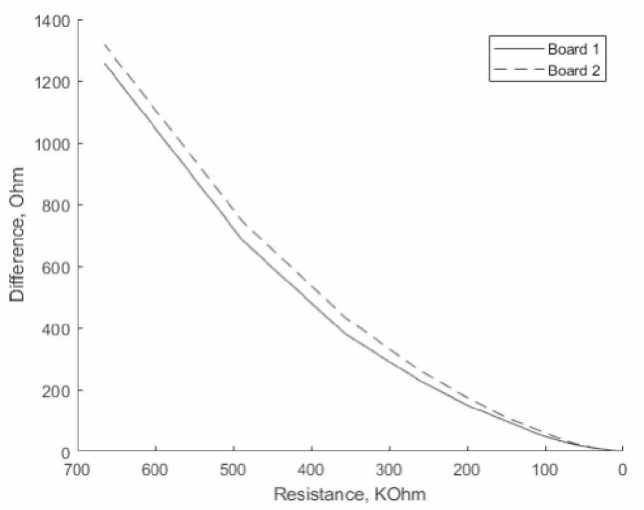

Figure 5-7. Difference between resistances measured with the Logger at 10 SPS and the reference ohmmeter

The curves that represent the difference between average resistances measured with Board 1 at 5 SPS and 10 SPS and with the reference ohmmeter are used as calibration curves. Each measured resistance is interpolated (if it lies within the range of 665 kΩ to 5 kΩ) or extrapolated (if it falls outside the range) on one of the curves depending on the set data rate. This gives the value to be subtracted from the measured resistance to get a calibrated result. Ideally, the calibration curves should have been

determined for all data rates since the input-referred noise depends heavily on the data rate value. However, due to the presence of the 60-Hz noise, this could not be done.

To verify the calibration, four resistors (specifications are presented in Table 5-6) were measured with the reference ohmmeter and Board 1 at 5 SPS (1700 samples were taken for each resistor). During the measurements, ambient temperature fluctuated within 0.5° C from 23.0° to 23.5° C.

Table 5-6. Specifications of the resistors for calibration verification

Resistor Model	Type	Nominal Resistance, kΩ	Tolerance, %	Temperature Coefficient, ppm/°C	Represented temperature for 20K thermistor, °C	Represented temperature for 10K thermistor, °C
Y0011475K000B0L	TH	475	±0.1	±2.5	-34.65	-45.10
9C08052A1243FKHFT	SMT	124	±1	±100	-12.10	-24.15
RMCF0603FT47K0	SMT	47	±1	±100	6.55	-7.00
RC0603FR-0733KL	SMT	33	±1	±100	13.95	-0.20

Measurement results are presented in Table 5-7. Drift was observed only for the 124 kΩ resistor and the drift value of 2.5 Ω agreed with the resistor's TCR of 100 ppm/°C. To obtain average resistances, 510 drift- and outlier-free samples from each measurement were used. Measurement results showed that calibration considerably improved the accuracy, especially for larger resistances.

Table 5-7. Calibration verification results

Resistor ID	Ideal resistance, Ω	Average resistance measured with Board 1, Ω	Difference between uncalibrated and ideal resistances, Ω	Calibrated average resistance measured with Board 1, Ω	Difference between calibrated and ideal resistances, Ω
475K	475037.00	475675.13	638.13	475042.82	5.82
124K	123175.00	123241.48	66.48	123173.30	-1.70
47K	47069.10	47081.52	12.42	47068.24	-0.86
33K	32999.00	33008.61	9.61	33000.95	1.95

5.2.1.2 Deriving the Equivalent Temperature Accuracy of the Logger

Measured resistances were converted to temperature using PCHIP interpolation onto resistance-temperature curves of 10 and 20 kΩ thermistors. Next, the equivalent temperature accuracy was calculated as the difference between temperatures obtained with the Logger and the reference

ohmmeter. As mentioned in 5.2.1.1, for Board 2, 5000 samples at a data rate of 10 SPS, and for Board 1, 510 samples at a data rate of 10 SPS and 5 SPS were used to obtain the statistical data. The uncalibrated data are presented in Table E-4 through Table E-6 in Appendix E.

Figure 5-8 presents the calculated equivalent temperature accuracy for Board 2 at 10 SPS with 10 kΩ (left) and 20 kΩ (right) thermistors. In the figure, dots denote the minimum, the average, and the maximum values. The difference between the maximum and the minimum values is the temperature accuracy range. As seen from the figure, the temperature accuracy becomes worse than 0.01° C at temperatures below -11.5° C for the 10 kΩ thermistor (Figure 5-8.(left)), and below 1.5° C for the 20 kΩ thermistor (Figure 5-8.(right)). Therefore, if an application requires the temperature accuracy to be less than 0.01° C in a wide temperature range, the 10 kΩ thermistor is the optimal choice. To compare the average temperature accuracies for 10 kΩ and 20 kΩ thermistors, both are plotted on the same graph (Figure 5-9).

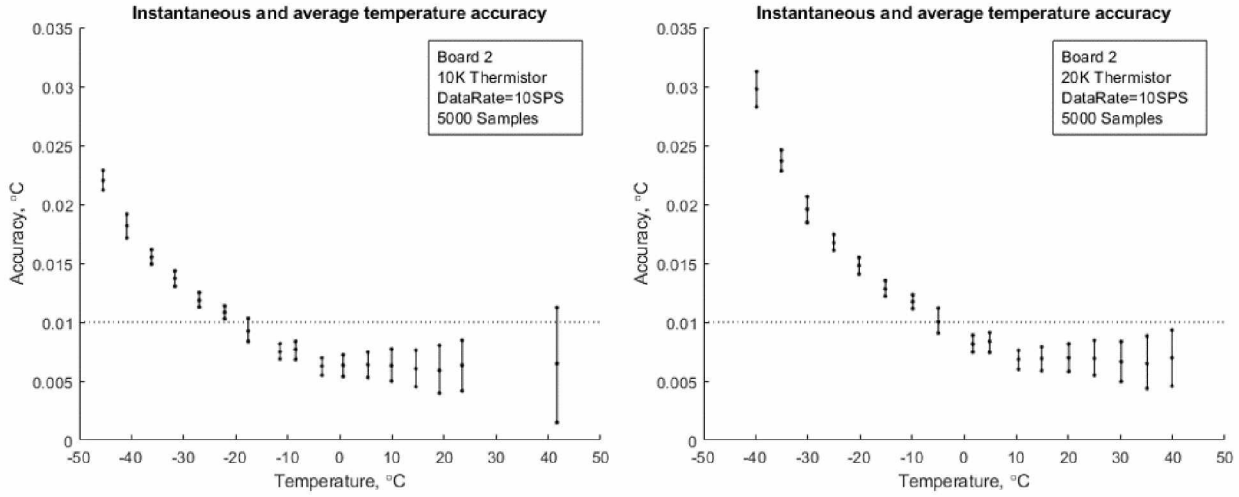

Figure 5-8. Board 2: the instantaneous and the average temperature accuracy for 10 kΩ (left) and 20 kΩ (right) thermistors at 10 SPS

The measured temperature resolution was calculated using 2.13, with V_{NOISE} being a peak-to-peak variation around an average measured voltage and $dV_{1°C}$ being a theoretical input voltage resolution required to achieve 1° C temperature resolution. The latter was used because the measured data could not provide enough points to calculate this parameter. Even though the measured voltage was slightly larger than the theoretical voltage, it is still appropriate to use the theoretical voltage to calculate the measured temperature resolution, since 2.13 uses the voltage difference $dV_{1°C}$ and not the absolute value of the voltage. Figure 5-10 depicts the measured and the expected temperature

resolution for Board 2 at 10 SPS. As seen from the figure, the measured resolution is worse than the expected resolution. This is explained by the measured noise voltage being higher than expected, as discussed in 5.2.1.1.

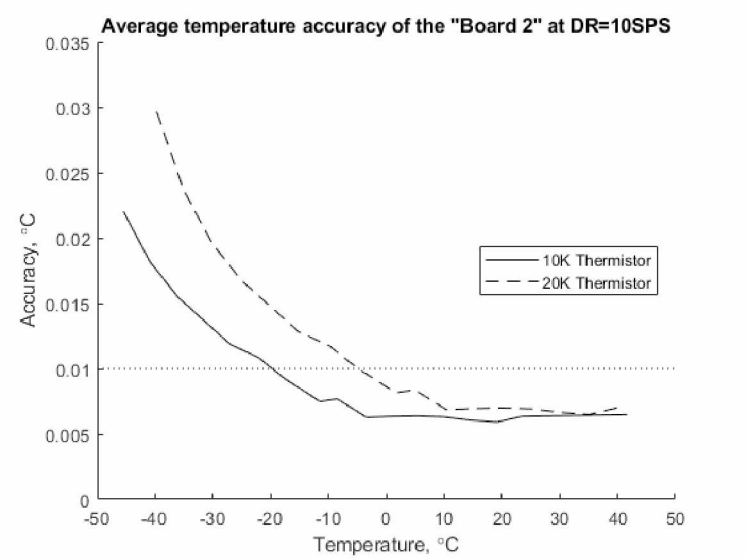

Figure 5-9. Board 2: the average accuracies for 10 kΩ and 20 kΩ thermistors

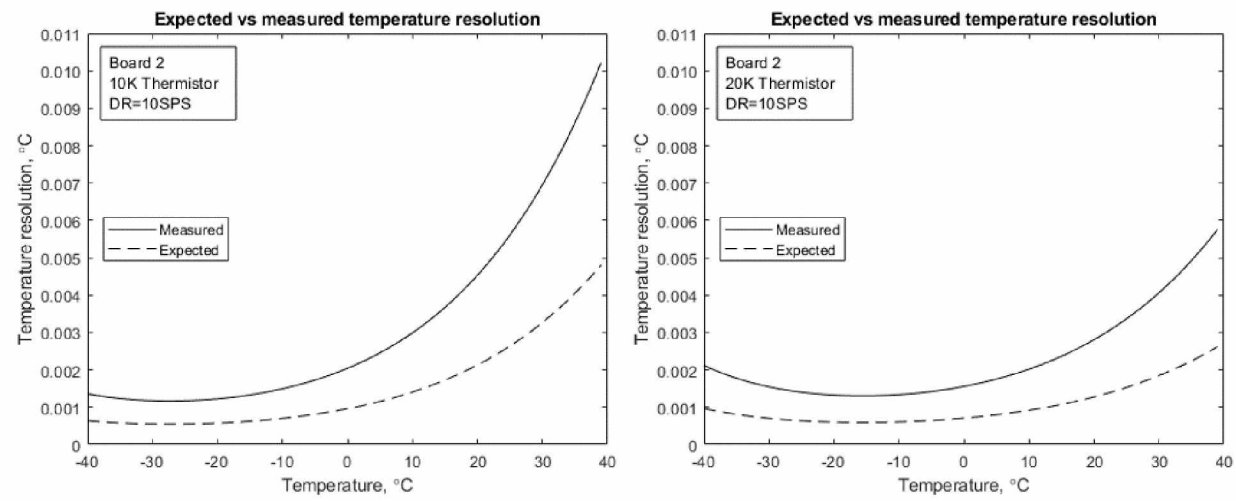

Figure 5-10. Board 2: the expected and the measured temperature resolutions for 10 kΩ and 20 kΩ thermistors at 10 SPS

Temperature accuracy and resolution are related to each other — at higher temperatures, the accuracy range increases because of the decrease in the resolution. This is illustrated for 10 kΩ and 20 kΩ thermistors in Figure 5-8: at temperatures above 10° C, the accuracy range becomes larger.

Therefore, if an application requires better temperature resolution at positive temperatures, the 20 kΩ thermistor is the right option.

Three plots similar to the ones shown above in this subsection are generated for Board 1 operating at a data rate of 5 SPS.

As mentioned in 5.2.1.1, Board 2 became unusable due to the ESD damage and no experiments were made with this board at a data rate of 5 SPS. The consequent experiments were performed with Board 1 operating at a data rate of 10 SPS and 5 SPS. Measurement results for Board 1 at 10 SPS were similar to the results obtained for Board 2 at the same data rate. Therefore, they are not covered in this section. The uncalibrated temperature accuracy data for this case can be found in Appendix E. On the other hand, the 5 SPS resistance data was better than the 10 SPS data for both boards, making it a more attractive subject to study. Three plots similar to the ones for Board 2 at 10 SPS were generated for Board 1 at 5 SPS. Figure 5-11 presents the equivalent temperature accuracy calculated for 10 kΩ and 20 kΩ thermistors, Figure 5-12 illustrates the comparison of the average accuracies for these cases, and Figure 5-13 shows the measured and the expected temperature resolution.

As seen in Figure 5-11, the temperature accuracy becomes worse than 0.01° C at temperatures below -22° C for the 10 kΩ thermistor (Figure 5-11.(left)), and below -9.9° C for the 20 kΩ thermistor (Figure 5-11.(right)). This result is 10° C better on average than for both boards at 10 SPS.

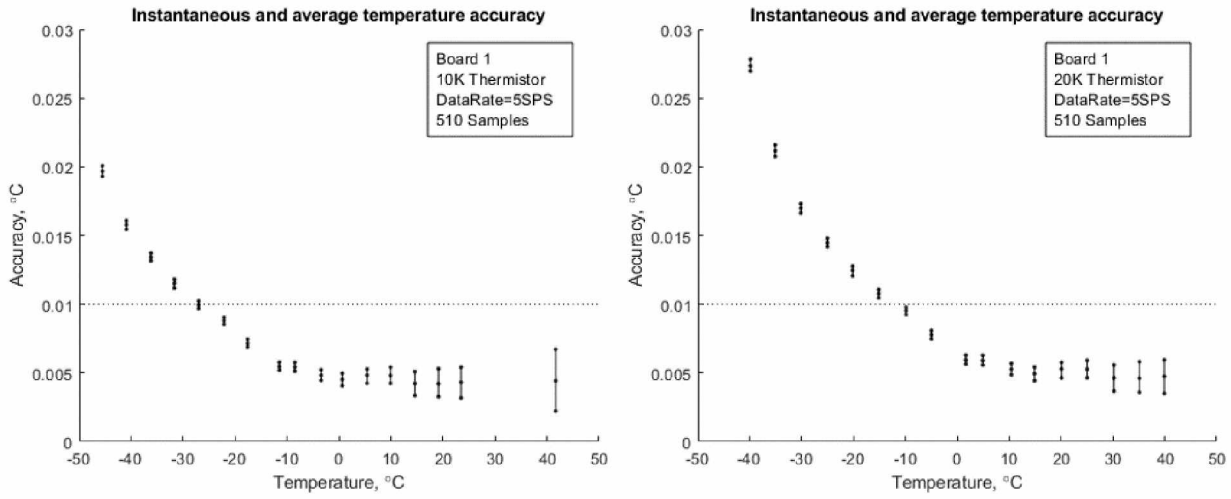

Figure 5-11. Board 1: the instantaneous and the average temperature accuracy for 10 kΩ (left) and 20 kΩ (right) thermistors at 5 SPS

Calculated temperature resolution for Board 1 at 5 SPS is very close to the theoretical result presented in 2.1.6 (Figure 5-13). This is explained by the fact that at 5 SPS measured noise voltage is very

close to the specified input-referred noise voltage of the ADS1247, which was used in calculations in 2.1.6.

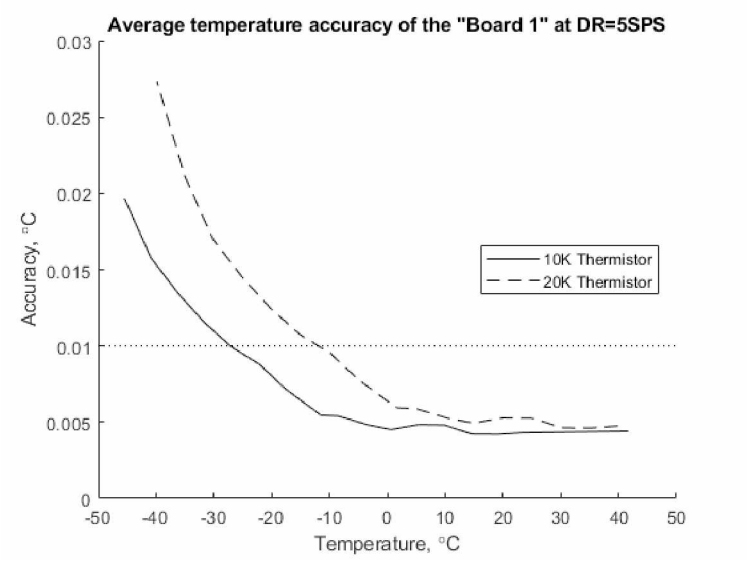

Figure 5-12. Board 1: comparison of the average temperature accuracies for 10 kΩ and 20 kΩ thermistors at 5 SPS

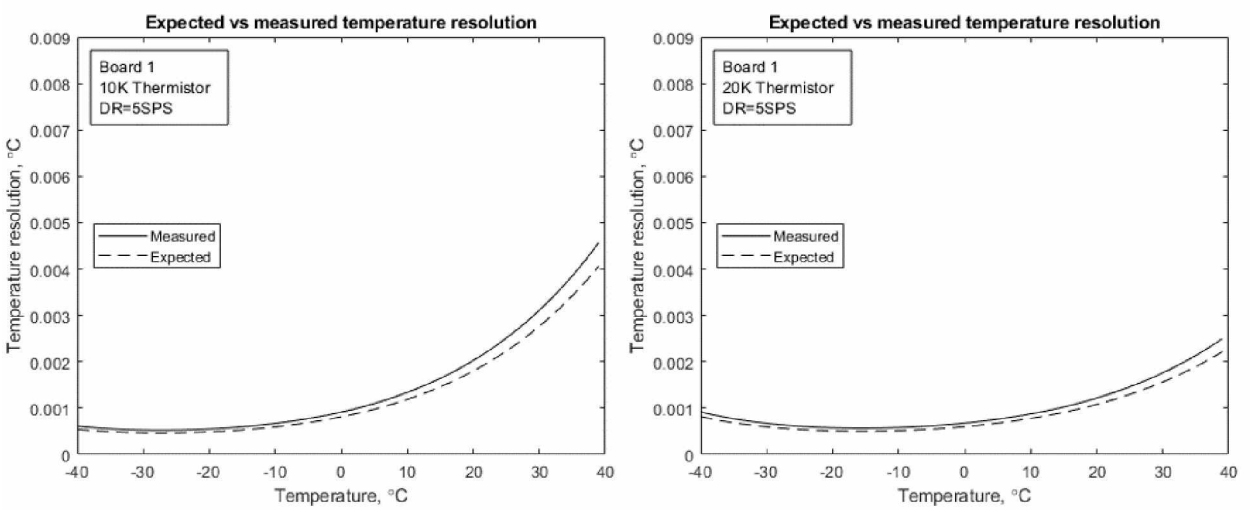

Figure 5-13. Board 1: expected and measured temperature resolutions for 10 kΩ and 20 kΩ thermistors at 5 SPS

The temperature resolution for both boards at data rates of 10 SPS and 5 SPS is better than 0.01° C within the temperature range from -40° C to 40° C. The only exception is the temperature resolution of 0.01021° C at 40° C for Board 2 at 10 SPS with the 10 kΩ thermistor (Figure 5-10). This proves that the design goal for a 0.01° C temperature resolution is achieved.

As specified in Table 5-1, the temperature of 0° C is approximated with the 32 kΩ resistor for the 10 kΩ thermistor and with the 60 kΩ resistor for the 20 kΩ thermistor. Regardless of the board or the data rate, the equivalent temperature accuracy for these resistors is smaller than 0.01° C, meaning that the design goal is successfully achieved.

To verify the developed calibration technique four additional resistors were measured with Board 1 at 5 SPS. The measurement results are presented in Section 5.2.1.1 (Table 5-7). These results were used to calculate the uncalibrated and calibrated equivalent temperature accuracies for both 10 kΩ and 20 kΩ thermistors (Table 5-8, Figure 5-14).

Table 5-8. The equivalent uncalibrated and calibrated temperature accuracies

Resistor ID	10 kΩ thermistor		20 kΩ thermistor	
	Uncalibrated equivalent accuracy for Board 1, °C	Calibrated equivalent accuracy for Board 1, °C	Uncalibrated equivalent accuracy for Board 1, °C	Calibrated equivalent accuracy for Board 1, °C
475K	0.0194765	0.0001778	0.0209460	0.0001912
124K	0.0090456	-0.0002314	0.0097893	-0.0002504
47K	0.0049402	-0.0003421	0.0053810	-0.0003727

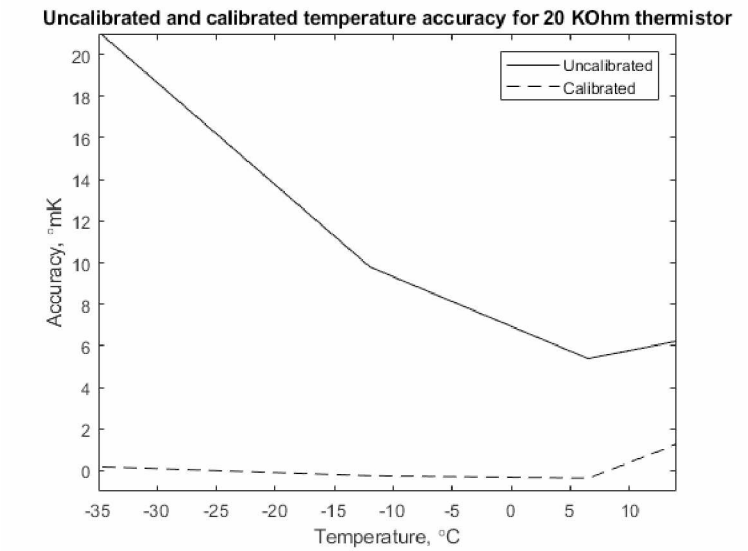

Figure 5-14. The equivalent uncalibrated and calibrated temperature accuracies for the 20 kΩ thermistor

5.2.1.3 Resistance Measurements with the CR1000

Resistance measurements for the CR1000 were made with the same 18 resistors as discussed in 5.2.1 (Table 5-1). Each resistor was first measured with the reference ohmmeter, then with the Logger, and immediately after, with the CR1000. During each measurement time series, the room's ambient

temperature was monitored using Hobo UX120-006M. Temperature fluctuations fell within 0.5° C for each series making the associated errors negligible. For the CR1000, up to 11000 samples were taken for each resistor, from which 5000 samples were used to obtain the statistical data.

To evaluate the results, it is necessary to know the CR1000 specifications [51] pertaining to the half-bridge scheme with single-ended input voltages. These specifications are provided in Table 5-9.

The CR1000 outputs a value that is the ratio of the measured voltage divided by the excitation voltage. The measured voltage and resistance are derived from this value using Eq. 5.1 and Eq. 5.2 respectively:

$$V_{MEAS} = X * V_{REF} \tag{5.1}$$

$$R_{MEAS} = R_{REF} * \frac{X}{1 - X}, \tag{5.2}$$

where V_{MEAS} is the measured voltage, X is the CR1000's output, V_{REF} is the reference voltage, R_{MEAS} is the measured resistance, R_{REF} is the reference resistance.

Table 5-9. Specifications of the CR1000 pertaining to the half-bridge scheme with single-ended input voltages

Parameter	Value
Resolution, Bit	13
Resolution, µV	667
Accuracy, V	± (0.06% of reading + offset), 0° to 40° C ± (0.12% of reading + offset), -25° to 50° C ± (0.18% of reading + offset), -55° to 85°C Offset is 3·resolutions + 3.0 µV
Reference voltage, V	2.5
Reference voltage accuracy, V	± (0.06% of setting + 0.8 mV), 0° to 40°C ± (0.12% of setting + 0.8 mV), -25° to 50°C ± (0.18% of setting + 0.8 mV), -55° to 85°C

The CR1000's program uses Eq. 5.1 and Eq. 5.2 to calculate the measured voltage and resistance and stores it in the memory.

In the experiment, measured voltages were provided by the CR1000 as either three distinct values spaced at a half-resolution voltage, or as a single value. The measured resolution was 680 µV, which is slightly larger than the specified value. Similarly, the measured resistances were provided as either three values or a single value; an example for the 147 kΩ resistor is shown in Figure 5-15. Out of

the three values, some occurred more frequently, and some occurred less frequently. The actual counts are displayed in Table 5-10 along with the measured resistance values.

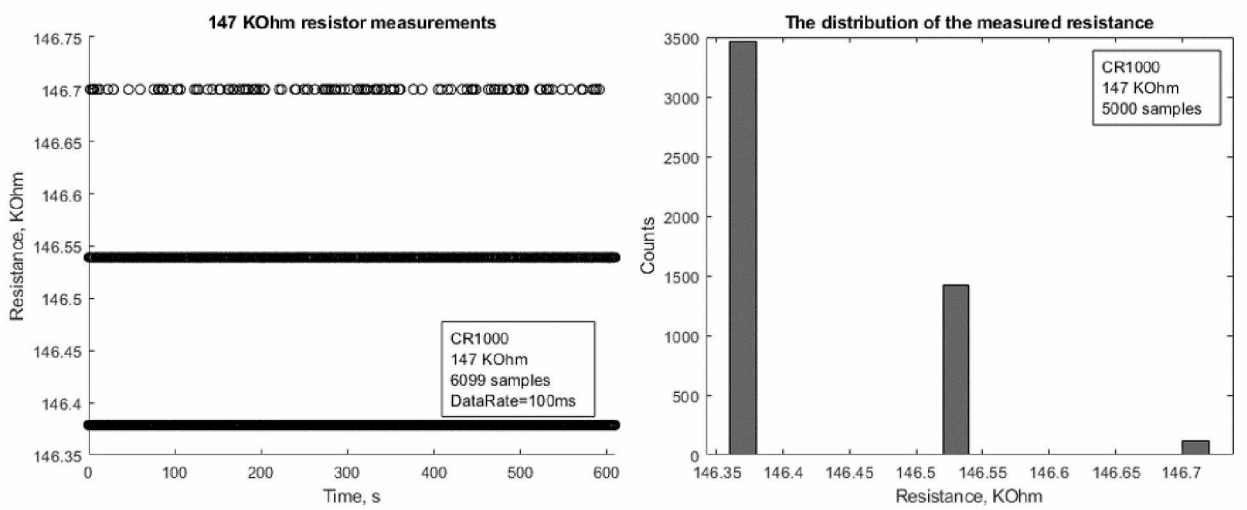

Figure 5-15. Measured resistance values and their distribution for the 147 kΩ resistor

Table 5-10. The CR1000 resistance measurement results

Resistor ID	Ideal resistance, Ω	Measured resistance, Ω			Difference between the most frequent value and ideal resistance, Ω / %
		The most frequently occurring value / the number of occurrences	Less frequently occurring value / the number of occurrences	The least frequently occurring value / the number of occurrences	
665K	665405.00	664349.40 / 3506	661760.70 / 1358	659191.30 /82	1055.60 / 0.1586
487K	487418.00	486762.70 / 1989	485344.80 / 10	483924.00 / 1	655.30 / 0.1344
357K	356756.00	354937.30 / 3382	355728.10 / 1434	356522.60 / 184	1818.70 / 0.5098
261K	261052.00	260163.40 / 5000	-	-	888.60 / 0.3404
196K	196078.00	195536.50 / 7238	195006.10 / 18	195270.90 / 1	541.50 / 0.2762
147K	147032.00	146378.50 / 3460	146539.10 / 1420	146700.10 / 120	653.50 / 0.4445
110K	109802.00	109386.20 / 5000	-	-	415.80 / 0.3787
85K	84516.00	84151.17 / 2429	84086.16 / 1344	84216.29 / 1227	364.83 / 0.4317
60K	60001.50	59757.67 / 5000	-	-	243.83 / 0.4064
51K	50998.90	50784.54 / 5000	-	-	214.36 / 0.4203
39K	39001.50	38809.02 / 5000	-	-	192.48 / 0.4935
32K	31597.60	31462.32 / 5000	-	-	135.28 / 0.4281
25K	24894.30	24783.25 / 5000	-	-	111.05 / 0.4461
20K	19997.80	19911.45 / 5000	-	-	86.35 / 0.4318
16K	16002.00	15927.99 / 5000	-	-	74.01 / 0.4625
13K	13001.10	12936.40 / 5000	-	-	64.70 / 0.4977
11K	10701.60	10654.99 / 5000	-	-	46.61 / 0.4355
5K	4982.33	4952.96 / 5000	-	-	29.37 / 0.5896

Measurements with large resistors were affected by a unique process that is illustrated in Figure 5-16. The figure shows that measured values did not settle until some time after the measurement start. This behavior was noted for the 665 kΩ, the 487 kΩ, and the 387 kΩ resistors. Because of this, only 2000 samples were used to obtain the statisitcal data for the 487 kΩ resistor.

To evaluate the measurement resuslts, I estimated the voltage error based on the specified voltage inaccuracy (Table 5-9). I then converted this error into the resistance inaccuracy and compared the estimated and the measured resistance errors.

Acknowledging that there are many sources of error (e.g., a reference voltage error, a reference resistance error, the loading error due to the ADC's finite resistance, etc.), an approximate error estimation for the inaccuracy of the CR1000 alone was made by not taking into account other sources of error and using nominal values of the parameters involved in the calculation.

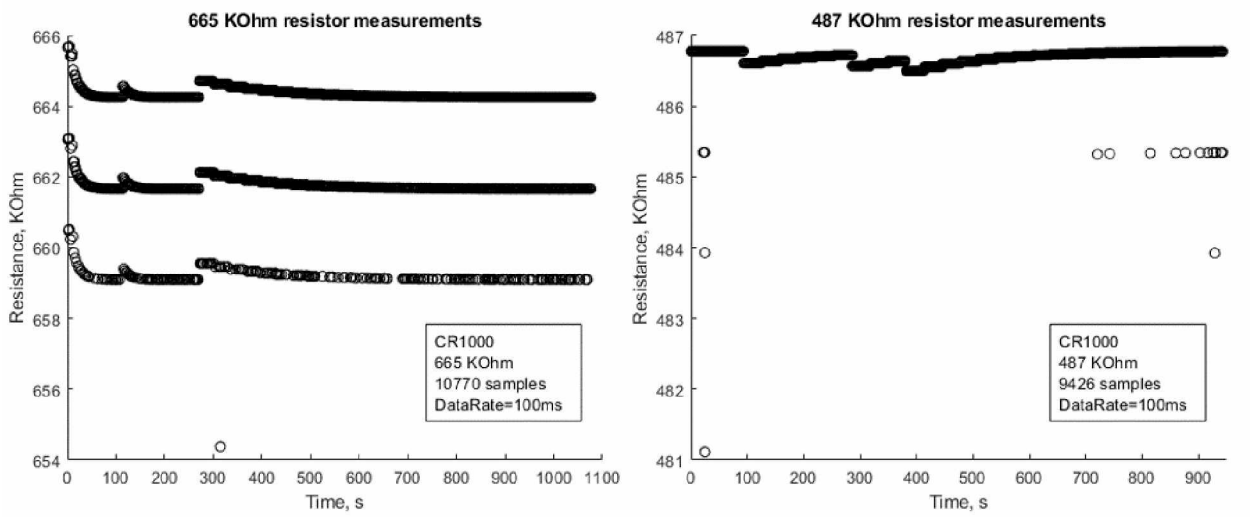

Figure 5-16. Measured resistances for the 665 kΩ and the 487 kΩ resistors

At room temperature, the CR1000 voltage error V_{ERROR} is specified as 0.06% of the measured voltage plus offset (Table 5-9). The voltage appearing at the CR1000 input can be considered as the true voltage V_{TRUE} of the half-bridge scheme. It can be found using Eq. 5.3 below:

$$V_{TRUE} = \frac{V_{REF} * R_{TRUE}}{R_{REF} + R_{TRUE}} \qquad 5.3$$

where R_{TRUE} is the true value of the resistance to be measured by the CR1000.

The voltage reported by the CR1000, which I call the estimated voltage V_{EST}, is the sum of the true voltage V_{TRUE} and the voltage error V_{ERROR}.

The true resistance can be found from the true voltage using Eq. 5.4, which is derived from Eq. 5.3 above.

$$R_{TRUE} = R_{REF} * \frac{V_{TRUE}}{V_{REF} - V_{TRUE}} \quad 5.4$$

Using the same equation, the estimated resistance, which is the resistance reported by the CR1000, can be determined as:

$$R_{EST} = R_{REF} * \frac{(V_{TRUE} \pm V_{ERROR})}{V_{REF} - (V_{TRUE} \pm V_{ERROR})} \quad 5.5$$

The estimated resistance error is then calculated from the difference between R_{EST} and R_{TRUE}. An example calculation made with equations 5.3 - 5.5 for the 11 kΩ resistor yielded an estimated error of ±50 Ω. The difference between resistances measured with the reference ohmmeter and the CR1000 is 46.61 Ω (Table 5-10), which is very close to the estimated value of ±50 Ω. Using this same method, the calculations were repeated for all other resistors. The estimated results agreed with the experimental results for resistors below 60 kΩ. However, for resistors above 60 kΩ, the estimated results were larger, and in case of the 487 kΩ and the 665 kΩ significantly larger (x10 times), than the experimental results.

5.2.1.4 Deriving the Equivalent Temperature Accuracy for the CR1000

For each resistor, the most frequently occurring value was converted into temperature °C using a PCHIP interpolation. The equivalent temperature accuracy was then calculated as the difference between temperatures obtained with the CR1000 and the reference ohmmeter. The CR1000's temperature accuracies for 10 kΩ and 20 kΩ thermistors are shown in Figure 5-17.

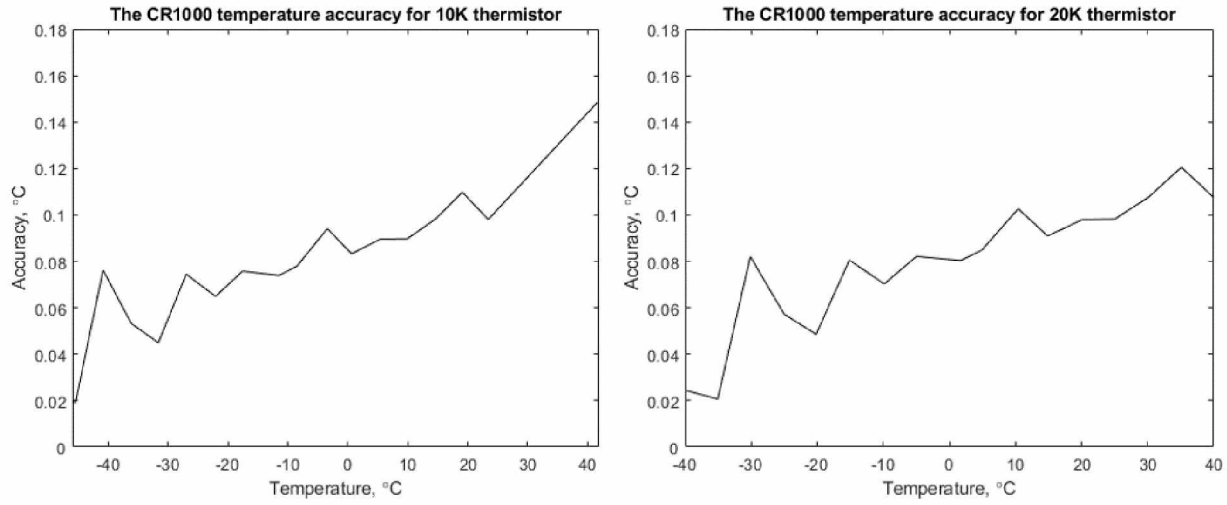

Figure 5-17. The CR1000 equivalent temperature accuracy for 10 kΩ (left) and 20 kΩ (right) thermistors

As seen from Figure 5-17, the equivalent accuracy is worse than 0.02° C in the entire temperature range for both thermistor types. This result can be easily explained by looking at the CR1000 specifications. For the half-bridge schematic with the 2.5 VDC voltage reference and the 25 kΩ reference resistor, the minimum voltage resolution required to achieve the 0.01° C temperature resolution is 116 µV for the 10 kΩ thermistor and 59 µV for the 20 kΩ thermistor (Figure 5-18.(left)). However, the CR1000's voltage resolution is 667 µV, which is insufficient to provide the required resolution and accuracy. Because the measured voltage resolution of 680 µV is very close to the specified resolution of 667 µV, the measured temperature resolution is also very close to the expected temperature resolution shown in Figure 5-18.(right).

To improve resolution and accuracy, GIPL uses an intermediate averaging of measured data: for an hourly measurement rate, the CR1000 loggers measure temperature every 3 minutes, and then average 20 samples to provide an hourly output. The biggest drawback of this method is an increased power consumption.

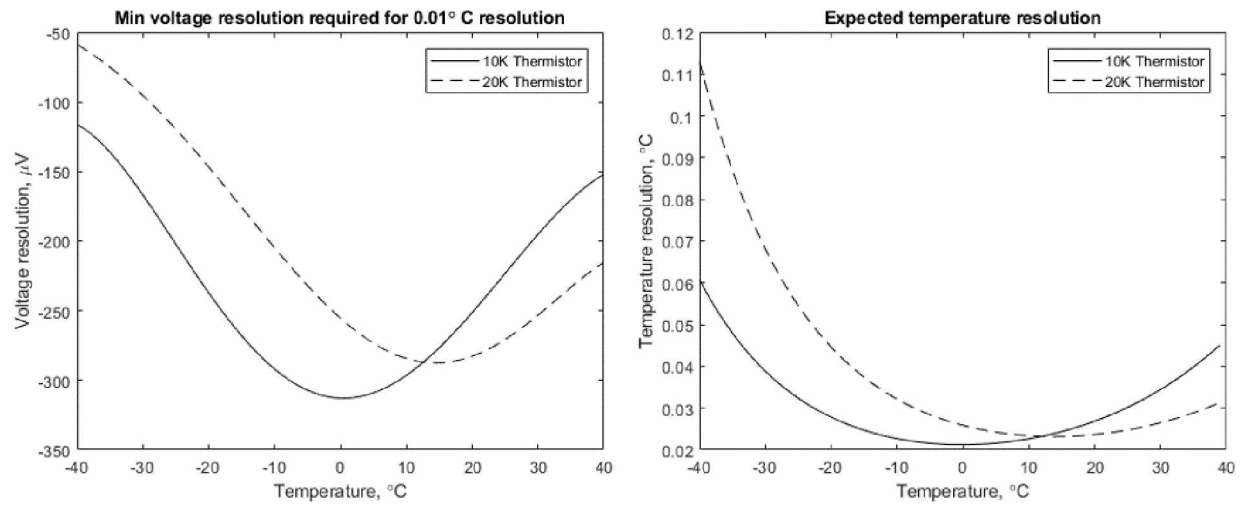

Figure 5-18. The minimum voltage resolution required for a 0.01 °C temperature resolution (left) and the expected temperature resolution (right) for 10 kΩ and 20 kΩ thermistors

5.2.1.5 Comparison of the Logger and the CR1000 when Measuring Resistors

The resistance accuracy and the equivalent temperature accuracy data for the Logger and the CR1000 were provided in previous sections of this Chapter. In this section, the two devices are compared directly. Table 5-11 summarizes the resistance measurements data.

Table 5-11. Resistances measured with the Logger and the CR1000

Resistor ID	Ideal resistance, Ω	Resistance measured with the Logger, Ω	Difference between the Logger and the ideal resistance, Ω / %	Resistance measured with the CR1000, Ω	Difference between the CR1000 and the ideal resistance, Ω / %
665K	665405.00	666614.74	1209.74 / 0.1818	664349.40	1055.60 / 0.1586
487K	487418.00	488081.34	663.34 / 0.1361	486762.70	655.30 / 0.1344
357K	356756.00	357132.71	376.71 / 0.1056	354937.30	1818.70 / 0.5098
261K	261052.00	261278.95	226.95 / 0.0869	260163.40	888.60 / 0.3404
196K	196078.00	196220.06	142.06 / 0.0725	195536.50	541.50 / 0.2762
147K	147032.00	147121.00	89.00 / 0.0605	146378.50	653.50 / 0.4445
110K	109802.00	109858.89	56.89 / 0.0518	109386.20	415.80 / 0.3787
85K	84516.00	84550.66	34.66 / 0.0410	84151.17	364.83 / 0.4317
60K	60001.50	60019.51	18.01 / 0.0300	59757.67	243.83 / 0.4064
51K	50998.90	51013.79	14.89 / 0.0292	50784.54	214.36 / 0.4203
39K	39001.50	39011.34	9.84 / 0.0252	38809.02	192.48 / 0.4935
32K	31597.60	31604.86	7.26 / 0.0230	31462.32	135.28 / 0.4281
25K	24894.30	24900.25	5.95 / 0.0239	24783.25	111.05 / 0.4461
20K	19997.80	20002.42	4.62 / 0.0231	19911.45	86.35 / 0.4318
16K	16002.00	16005.16	3.16 / 0.0197	15927.99	74.01 / 0.4625
13K	13001.10	13003.49	2.39 / 0.0184	12936.40	64.70 / 0.4977
11K	10701.60	10703.64	2.04 / 0.0191	10654.99	46.61 / 0.4355
5K	4982.33	4983.21	0.88 / 0.0177	4952.96	29.37 / 0.5896

Figure 5-19 illustrates the difference between the resistance values measured using the reference ohmmeter and both devices - the Logger (Board 1 at 5 SPS) and the CR1000. As seen from Figure 5-19, for almost the entire range of resistance values, the Logger's error is smaller than the CR1000's error. Applying the available calibration for the Logger, this difference becomes even more pronounced.

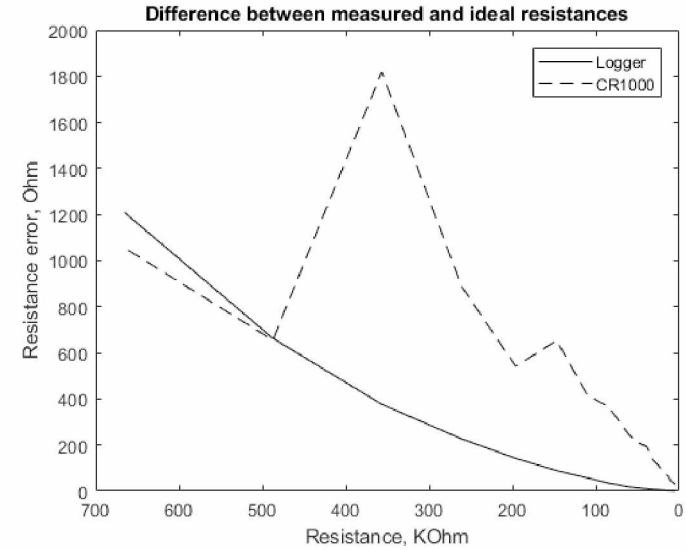

Figure 5-19. The resistance accuracy for the Logger and the CR1000

The uncalibrated equivalent accuraccy calculation results for the Logger (Board 1 at 5 SPS) and the CR1000 are presented in Table 5-12. The visual illustration of these data is presented in Figure 5-20.

Table 5-12. The equivalent temperature accuracy of the CR1000 and the Logger

Resistor ID	10 kΩ thermistor		20 kΩ thermistor	
	The CR1000 uncalibrated equivalent accuracy, °C	The Logger uncalibrated equivalent accuracy, °C	The CR1000 uncalibrated equivalent accuracy, °C	The Logger uncalibrated equivalent accuracy, °C
665K	-	-	0.024314161	0.0273463
487K	0.0192323	0.0196794	0.020681378	0.0211620
357K	0.0761614	0.0157738	0.082009166	0.0169847
261K	0.0530963	0.0134235	0.057252601	0.0144739
196K	0.0449537	0.0115364	0.048537168	0.0124563
147K	0.0744747	0.0099467	0.08052618	0.0107549
110K	0.0648737	0.0087965	0.070249225	0.0095254
85K	0.0757321	0.0071728	0.082119141	0.0077780
60K	0.0738771	0.0054623	0.080325101	0.0059391
51K	0.0779259	0.0054162	0.084832018	0.0058969
39K	0.0940373	0.0048299	0.102586264	0.0052683
32K	0.0832464	0.0045190	0.090907771	0.0049334
25K	0.0895151	0.0048369	0.097897805	0.0052888
20K	0.0896223	0.0048087	0.098124171	0.0052662
16K	0.0981598	0.0042266	0.107668597	0.0046353
13K	0.1097186	0.0042100	0.120488507	0.0046208
11K	0.0980403	0.0043161	0.107796291	0.0047472
5K	0.1484442	0.0044189	-	-

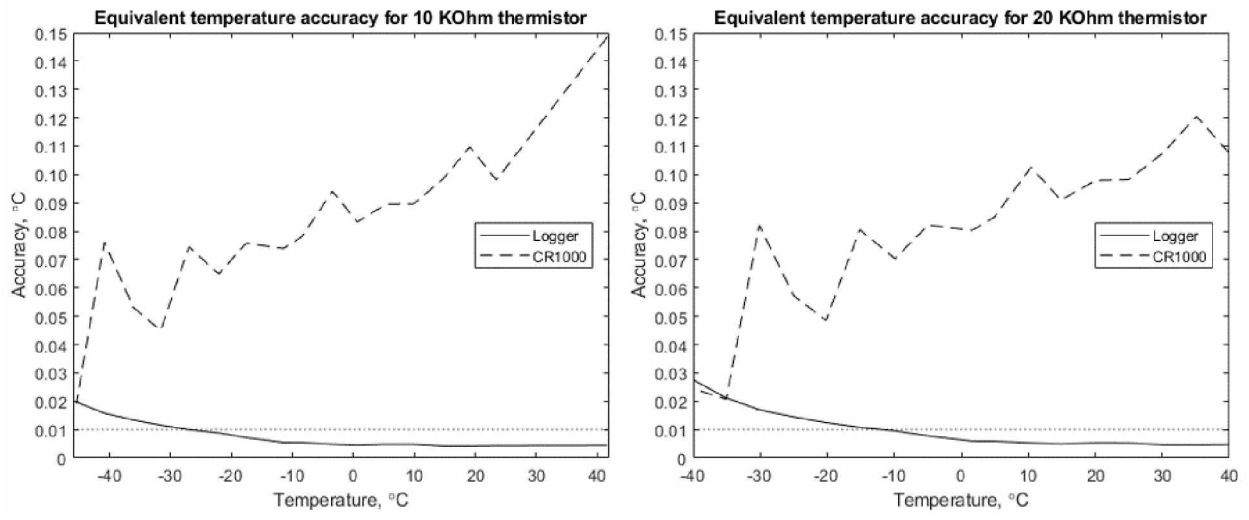

Figure 5-20. The uncalibrated equivalent temperature accuracy for the Logger and the CR1000: 10 kΩ (left) and 20 kΩ (right) thermistors

Similarly to the resistance data, the Logger demonstrates considerably better equivalent temperature accuracy than the CR1000 for both 10 kΩ (Figure 5-20.(left)) and 20 kΩ (Figure 5-20.(right)) thermistors. The two devices show comparable results only for the largest resistors (665 kΩ, 487 kΩ).

5.2.2 Measurements at Low Temperatures

To determine the effect of ambient temperature on measurement accuracy, a series of experiments were conducted where the Logger (Board 2) was measuring resistors while operating at different temperatures: -30°, -25°, -20° and 6° C. The results were then compared with the data obtained at room temperature.

A Forma Scientific, Inc. model 3676 freezer and a common refrigerator were used in the tests. To monitor and log temperature inside the freezer and the refrigerator, the Hobo UX120-006 was used with its sensor located near the Logger. The freezer's control knob indicates that its temperature can be set in the range from -20° to -35° C. However, measured temperatures were approximately 4° C higher than the set temperatures with the lowest value around -31.5° C. The refrigerator's temperature varied within the 1° C to 6° C range. I chose -30° C, -25° C, -20° C and 6° C for the experiments' temperatures based on the limits of the freezer and the refrigerator, as well as the available nominals of the resistors. I used eight resistors representing thermistors at these temperatures (Table 5-13).

Table 5-13. Resistors' values for the experiments at low temperature

Temperature, °C	Resistor value to match the 10 kΩ thermistor, kΩ	Resistor value to match the 20 kΩ thermistor, kΩ
-35°	487	261
-30°	357	196
-25°	261	147
-20°	196	110
6°	60	25

Ideally, Logger would be placed in the freezer and the resistors would be left outside, so only the Logger would be affected by low temperature. This would allow for the direct comparison of resistances measured at room and at low temperatures. However, two factors prevented me from doing this. First, air temperature in the room with the freezer was 4° C lower than in the room where the tests at room temperature were conducted. Second, to connect resistors, I needed at least 1.5 m long cables instead of the 0.15 m patch cords used in the previous experiments. Therefore, I would have to redo all tests at room temperature. Instead, I decided to place the Logger and the test bed with the resistors inside the freezer and measure the resistors with both the Logger and the reference ohmmeter. In this

case I could only compare the differences in the resistances measured with the Logger and the reference ohmmeter at room temperature and at low temperatures.

The Logger performs a self-calibration before the measurement start so for the calibration to be effective, the Logger would have to be cooled to approximately the same temperature as the intended experiment's temperature. To address this issue, I implemented the ad-hoc version of the firmware, which could delay the measurement start by 20 minutes, and thus would allow the Logger to cool down.

The experiments were conducted in the following order: (1) open the lid, connect a resistor to channel #2 of the Logger, press the button to start the delay timer, close the lid, (2) wait until the measurement is over, open the lid, disconnect the logger, connect the reference ohmmeter leads to the resistor, close the lid, (3) measure resistor with the reference ohmmeter, (4) repeat steps 1 – 3. Prior to all experiments, I placed the Logger, the test bed, and the reference ohmmeter leads into the freezer and left them there for 30 minutes to cool down. The test bed with resistors and the reference ohmmeter leads were not taken out of the freezer until the experiments had ended while the Logger was sometimes removed from the freezer to read out data and check the battery voltage.

To minimize the error when calculating the difference in resistances measured with the Logger and the reference ohmmeter, the two measurements had to be made at approximately the same temperature. For this, I selected periods when the freezer had reached the lowest temperature and turned off its compressor for both measurements. For temperature recording I used readings from the Hobo UX120-006 LCD screen, which were updated once every 10 seconds. I also made notes on temperatures and times for the following events when: (1) the freezer's lid was closed, (2) the freezer's compressor was turned on/off, (3) the measurement started, and (4) the measurement ended. When the experiments were finished, I used these notes and the logged readings from the Hobo UX120-006 to identify the periods of time for data acquisition (Figure 5-21.(left)).

Because of the delayed start and the measurement duration of 17 minutes (5100 samples at 5 SPS), I was able to obtain the periods when the freezer had reached its lowest temperature for all experiments. In all cases this low temperature was maintained within a 0.05° C interval for around 110-130 seconds. This was enough to acquire 510 drift-free samples (102 seconds at 5 SPS), which were used for further statistical analysis. An example of this 102-second interval can be seen in Figure 5-21.(right).

The measurement results are presented in Table 5-14. I identified a pattern that holds true for all resistors – the accuracy, which is defined as the difference between resistances measured with the reference ohmmeter and the Logger, improves as the operating temperature decreases. This is a very useful feature for the device, which will be operating mostly at cold temperatures. The downside of this

effect is that it would require determining new calibration curves for the calibration technique described in 5.2.1.1.

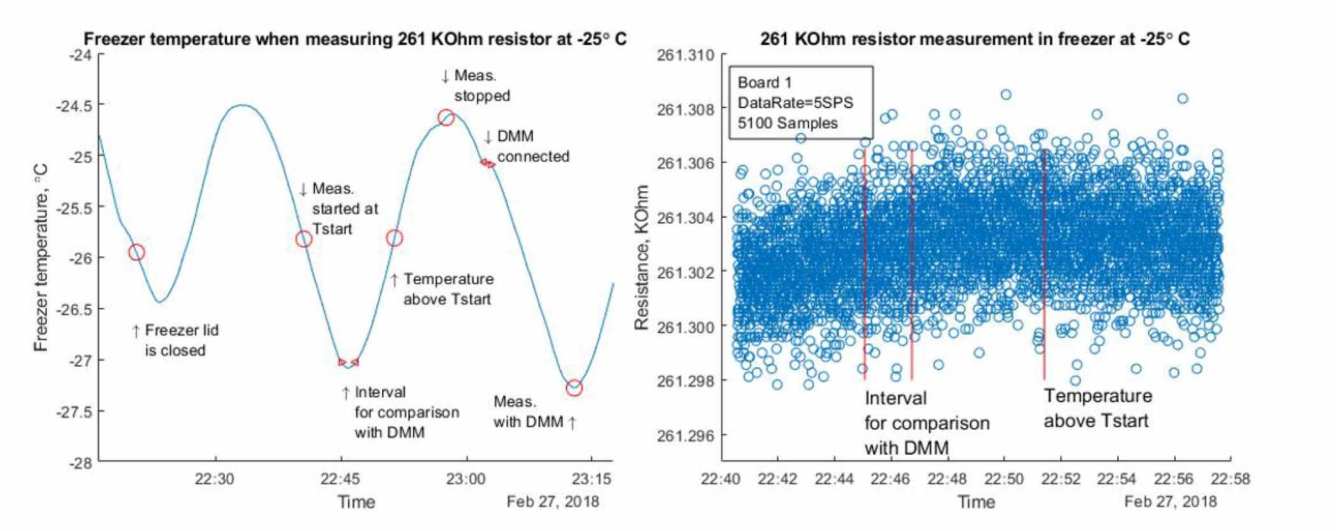

Figure 5-21. The example of a low temperature measurement

Table 5-14. Results of the measurements at low ambient temperatures

Resistor ID	Uncalibrated average resistance measured with Board 1, Ω	Ideal resistance, Ω	Difference between uncalibrated and ideal resistances, Ω	Conclusion
487K @ -30° C	487859.61	487294	565.61	Accuracy improves at lower temperature
487K @ 24° C	488081.34	487426	655.34	
357K @ -30° C	357014.67	356707	307.67	Accuracy improves at lower temperature
357K @ 24° C	357132.71	356767	365.71	
261K @ -30° C	261305.05	261137	168.05	Accuracy improves at lower temperature
261K @ -25° C	261301.85	261125	176.85	
261K @ 24° C	261278.95	261064	214.95	
196K @ -30° C	196249.13	196146	103.13	Accuracy improves at lower temperature
196K @ -20° C	196237.99	196128	109.99	
196K @ 24° C	196220.06	196090	130.06	
147K @ -25° C	147203.74	147147	56.74	Accuracy improves at lower temperature
147K @ 24° C	147121.00	147036	85	
109K @ -20° C	109862.34	109825.9	36.44	Accuracy improves at lower temperature
109K @ 24° C	109858.89	109810	48.89	
60K @ 6° C	60018.04	60000.3	17.74	Accuracy improves at lower temperature
60K @ 24° C	60019.51	60000.9	18.61	
51K @ 6° C	51010.24	50996.3	13.94	Accuracy improves at lower temperature
51K @ 24° C	51013.79	50999.2	14.59	
25K @ 6° C	24903.42	24897.6	5.82	Accuracy improves at lower temperature
25K @ 24° C	24900.25	24893.7	6.55	

5.3 The Ice-Bath Measurement

In the experiments described in 5.2, the Logger measured resistors and not temperature. To evaluate the Logger's ability to measure temperature, the ice-bath measurement was performed. The ice-bath is a simple yet very efficient way of obtaining a reasonably accurate 0 °C reference point (the ice point). If properly prepared, the uncertainty of the ice-water mixture is within ±0.005 °C [53] from 0 °C.

In the experiment, a single PS103J2 thermistor [23] connected to the second channel of Board 1 was used. The Logger operated in the RTC mode at a data rate of 5 SPS and a measurement rate of 12 seconds. The measurement result was calibrated using the technique described in 0. The ice-bath was prepared following the recommendations in [53]. The Logger was left in the ice-bath for 35 minutes. The measured temperature during this period is shown in Figure 5-22.

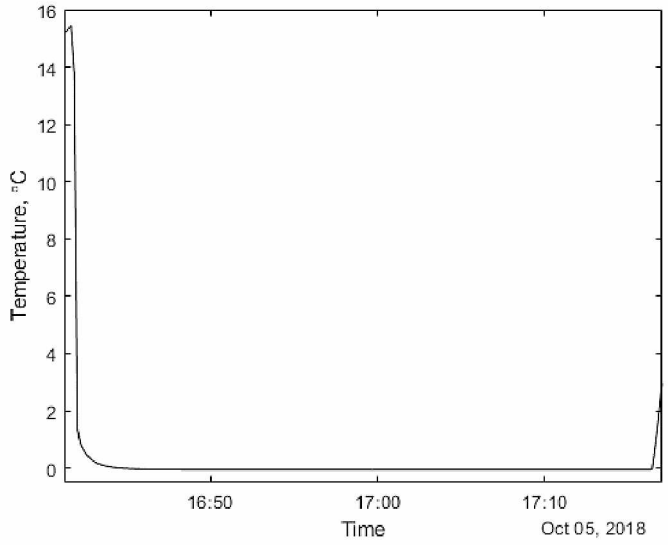

Figure 5-22. The ice-bath temperature measurement

The thermistor achieved equilibrium with the media 12 minutes after insertion into the ice-bath. The closeup view of the 22-minutes interval, when the thermistor was in equilibrium with the media, is presented in Figure 5-23. The average temperature was -0.0483° C. Temperature varied in the range of ±2.5 mK from the average value. Removing the two outliers, which occurred at 17:11:49 and 17:12:13, this number drops to ± 1.4 mK. These temperature variations correspond to the measured resistance variations of ± 4.27 Ω and ± 2.38 Ω respectively. The former is close to the results obtained in 5.2.

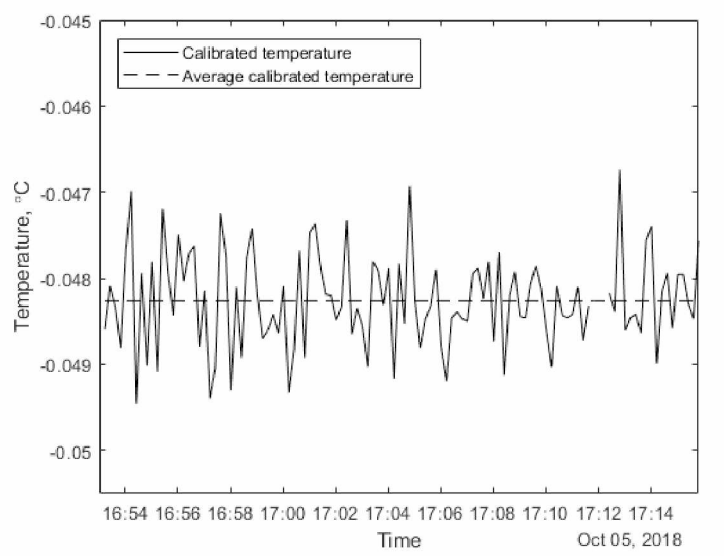

Figure 5-23. The thermistor temperature during equilibrium with media

The measured ice-bath temperature of -0.048° C is an acceptable result keeping in mind that the accuracy of the thermistor is ±0.1° C within the range 0° C to 70° C. For the Logger to demonstrate the required accuracy of ±0.01° C near 0° C and ±0.05° C elsewhere in the range from -40° C to 40° C when making temperature measurements with thermistors, it should undergo a multi-point calibration procedure. Additionally, it is preferable to use thermistors with ±0.05° C accuracy.

A calibrated reference thermometer and a constant-temperature bath, such as a Fluke 7040 [54], which has a stability of ±0.002° C at -40° C, ±0.0015° C at 25° C, and a setup step of 0.01° C are required for the multi-point calibration. The calibration procedure will consist of a series of tests conducted at different set temperatures of the bath. During these tests, the bath temperature will be measured with the Logger and monitored with the thermometer. To obtain a calibration function from the calibration procedure data, the approach as the one discussed in 5.2.1.1 will be utilized. The similar series of tests will then be conducted at different set temperatures of the bath to verify the accuracy of the Logger.

5.4 Air temperature measurements

Long-term outside experiments were conducted to check the real-life operation of the Logger and compare it with the CR1000. Initially, both devices stayed outside uncovered. Unfortunately, this resulted in a large amount of fluctuation in measurements. To improve the consistency in measurement readings, both devices were enclosed in the cooler box. The Logger and the CR1000's thermistors were located as close to each other as possible (Figure 5-24). The PS103J2 thermistors [23] were used in these experiments.

Figure 5-24. The test setup for outside measurements

All experiments took place from February 8 to February 23, 2018. The devices were subjected to a wide range of temperatures, from -30° C to 0° C. Measured temperatures for both devices were similar. Temperature curves for the devices only partially matched the air temperature data T_{Air} (Figure 5-25, Figure 5-27), collected at the GIPL "Smith Lake 1" site, located near Smith Lake on the UAF campus. This difference may have been caused by two main factors. First, "Smith Lake 1" site is located 27 meters lower than the experiment site (near 1060 N Chandalar Drive on UAF Campus). Second, the devices were in the closed cooler, while the temperature sensor at the "Smith Lake 1" site is located inside the specially designed radiation shield, which allows the outside air to ventilate the sensor [55].

A noticeable difference between the two devices is a much higher resolution of the Logger compared to the CR1000, a factor that results in smoother data curves. The close-up views are presented in Figure 5-25.(right) and Figure 5-27.(right). Higher resolution datasets become especially

important when measuring slowly changing temperatures. Also interesting to note is that the CR1000's temperature resolution is around 0.02° C at -3° C (Figure 5-27.(right)), which matches the expected temperature resolution shown in Figure 5-18.(right).

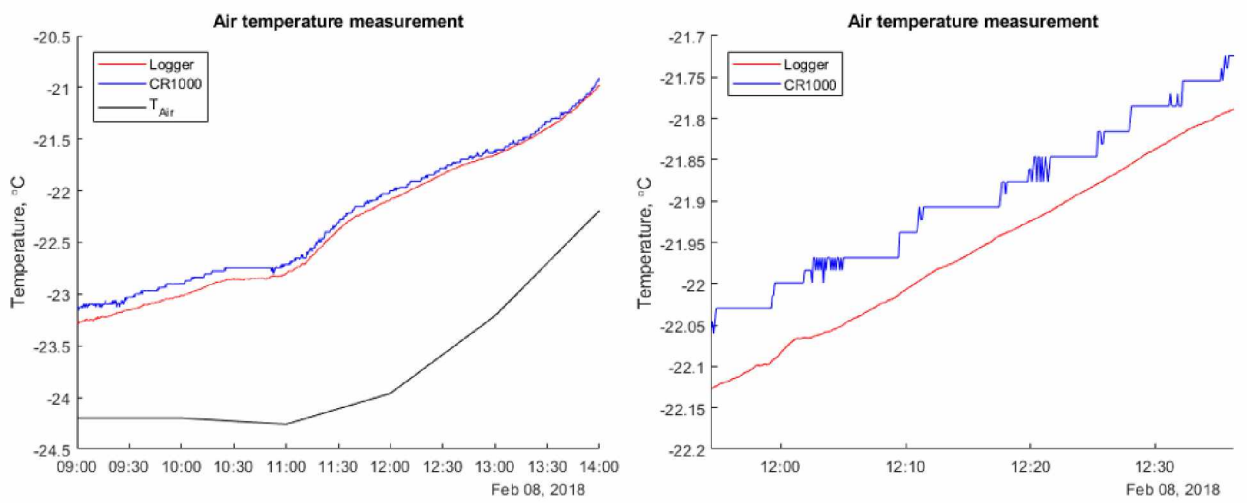

Figure 5-25. Air temperature measurement on Feb 8, 2018: 5-hours interval (left) and 50-minutes interval (right)

Figure 5-26 shows data from 16 channels' for both the Logger (right) and the CR1000 (left). The temperature, measured within this 3-minute interval, appears almost constant. This allowed me to determine the maximum channel-to-channel variation when using thermistors.

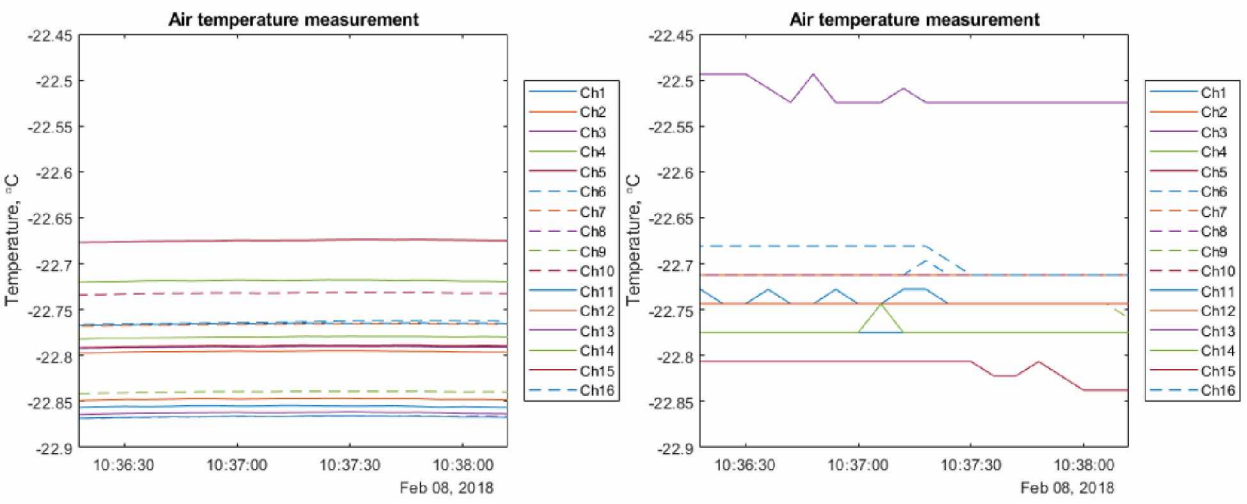

Figure 5-26. Air temperature measurement on Feb 8, 2018: 3-minute interval for the Logger (left) and the CR1000 (right)

For the logger, all values lie within 0.19° C (from -22.87° C to -22.68° C), which correlates with the thermistors' interchangeability of ±0.1° C. For the CR1000, the channel-to-channel variation is 0.32° C.

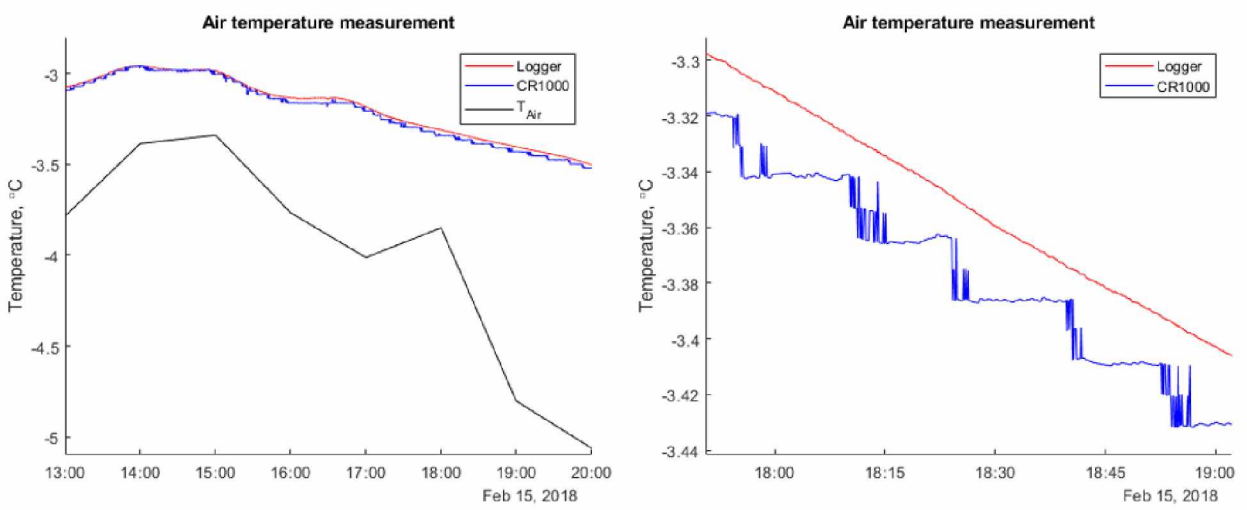

Figure 5-27. Air temperature measurement on Feb 15, 2018: 7-hour interval (left) and 35-minut interval (right)

5.5 Supply Current Measurements

A Keysight 66319D DC Mobile Source [56], which was set in the power supply mode, was used to measure the current drain of the Logger. The DC Mobile Source emulated a battery with a nominal voltage of 4.5 V and internal resistance of 0.1 Ω. Two series of measurements were made: at a minimal measurement rate for each data rate and at the fixed measurement rate of 12 seconds. The former was used to determine current drain and battery life for the measurement rate of 1 hour. During the measurements, 16 thermistors of the same type were connected to the Logger.

An example measurement session at a measurement rate of 12 s and a data rate of 5 SPS and 2000 SPS is shown in Figure 5-28. The current drain profile presented in the figure corresponds to the RTC mode of operation, which is discussed in 3.1.2.1.

The session begins when the Logger is disconnected from the host PC and starts measuring temperature. First, the ADC calibration and setup are performed followed immediately by the first measurement. The measurement interval is synchronized with the beginning of the session, so the second measurement starts shortly after the end of the first measurement. The first 10 measurements' data are written into the microcontroller's flash memory. Writing to flash is seen as a 6.3 mA peak.

Each consequent 10-measurements cycle does not contain the ADC calibration and setup. Instead, it starts with generating the real-time clock timestamp. This is seen as an 8.1 mA peak with an

approximate duration of 5 ms. Since the microcontroller wakes up during the time when RTC registers are updated, it must wait for the duration of the guard interval (4 ms) until it is safe to read from these registers [57]. For the very first cycle of the session, generating a timestamp doesn't produce such high current because the microcontroller does not need to wait until the guard interval expires. The cycle is finished with writing data to flash memory.

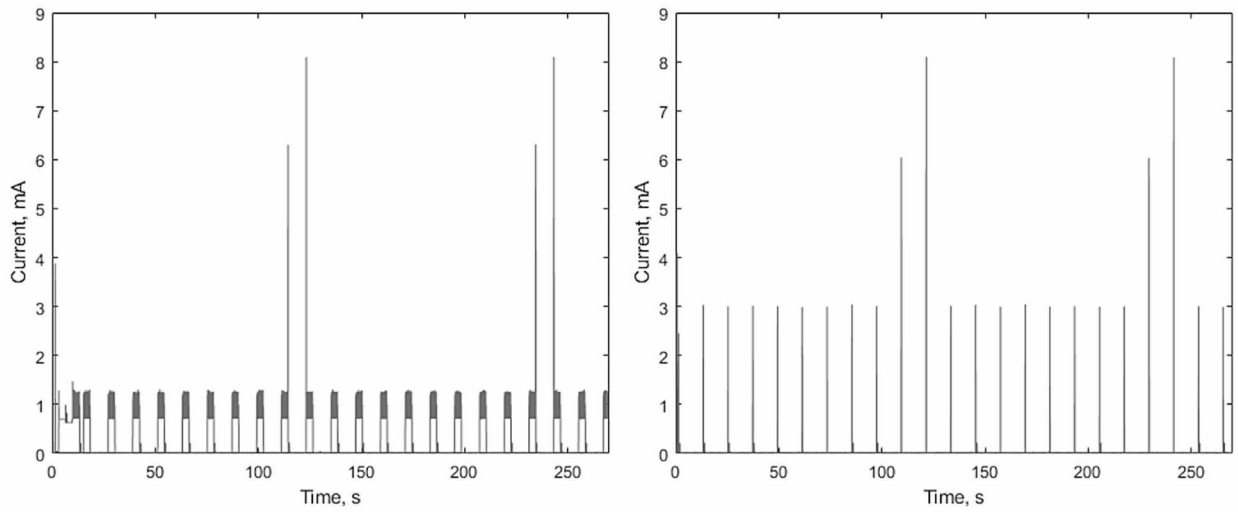

Figure 5-28. Current drain at a measurement rate of 12 s and a data rate of 5 SPS (left) and 2000 SPS (right)

Such 10-measurements cycle, as described above, is repeated 768 times and then the entire flash content is written to microSD-card. Writing to SD-card occurs so infrequently that it has very small effect on the average current drain of the Logger. The duration of the writing process is 725 ms, and the measured current during this process is 50 mA. This adds 3 µA to the average current of the Logger when it operates at the minimal measurement rate of 2 s, 0.5 µA when it operates at the measurement rate of 12 s, and 1.8 nA when it operates at the measurement rate of 1 hour. Therefore, current drain during the microSD-card writing was not accounted for when calculating the average current drain of the Logger. Also, the 10-measurement cycle was adopted as a basis for this calculation.

Measured and expected current drains at the minimal measurement rate are shown in Figure 5-29.(left). The minimal measurement rate is 12 s for a data rate of 5 SPS, 6 s for 10 SPS, 4 s for 20 SPS and 40 SPS, and 2 s for 80 – 2000 SPS. The difference between the measured and the expected current is around 30 µA for all data rates. This can be partially explained by the fact that the ADC's voltage regulator supplies some current to the thermistor, which was not accounted for when calculating

current drain in 2.2.2. The resistance of the 10 kΩ thermistor at room temperature is 10 kΩ. This causes extra 16 µA current to be drawn from the supply.

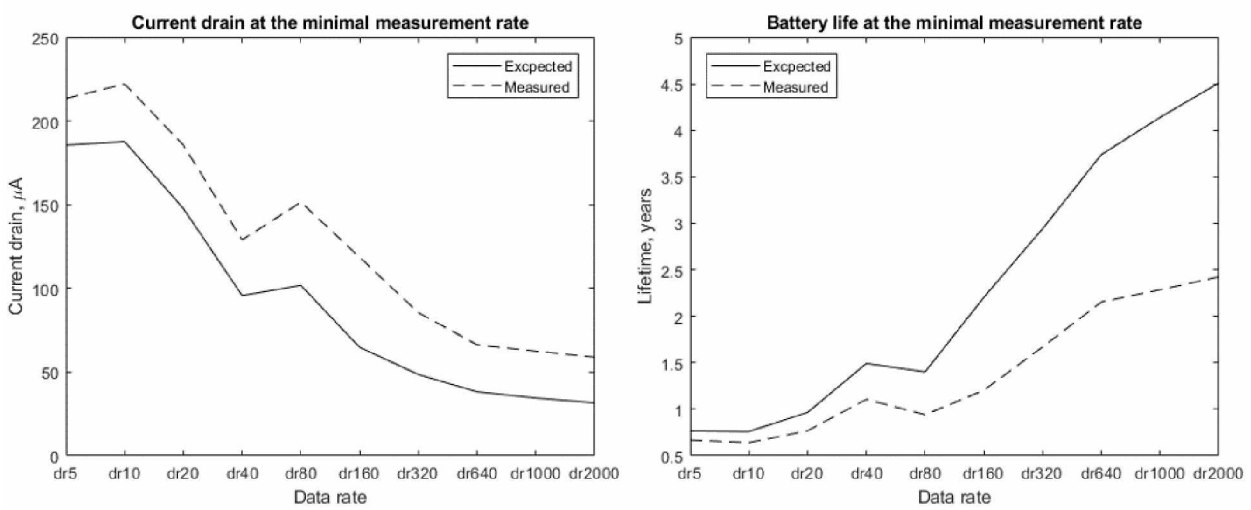

Figure 5-29. Current drain (left) and battery life (right) at the minimal measurement rate

Battery life for the case of the minimal measurement rate is shown in Figure 5-29.(right). As discussed in Section 2.2.2, battery life is determined by dividing the battery capacity in Coulomb by the measured average current.

Current drain and battery life at the measurement rate of 1 hour (Figure 5-30) were calculated by spreading the average current measured for the 120-seconds interval (one 10-measurements cycle at the measurement rate of 12 s) over a 10-hours interval.

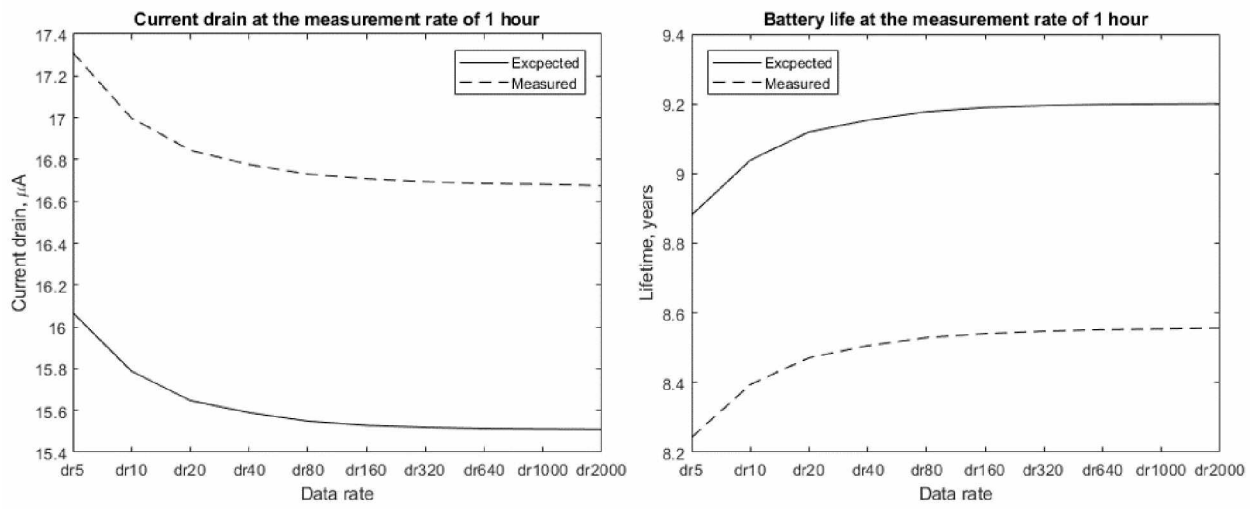

Figure 5-30. Current drain (left) and battery life (right) at the measurement rate of 1 hour

This spreading was achieved by separating the currents for two cases – the case when the Logger is active and the case when it sleeps in the low-power mode. The calculations results are presented in Figure 5-30. As seen from the figure, the estimated battery life is four times longer than the required battery life of two years.

Chapter 6 Conclusion and Future Work

This thesis presented the prototype of the data logger for precise soil temperature measurements that has 16 channels, can store up to 41.9 million measurements, is capable of withstanding low temperatures, and can be adapted for many applications. The Logger exceeds most of the design requirements and outperforms one of the best commercially available loggers in terms of temperature measurements. Despite the encouraging accomplishments, there are still many things to be implemented and improved.

There are several minor and major improvements to the schematic and the printed circuit board of the prototype that I would like to implement. The major improvements include the following:

1. Changing the sensing circuit schematic as described in 2.1.5.
2. Adding the anti-aliasing filter. This will allow me to determine calibration curves for data rates from 20 SPS to 2000 SPS while working in noisy lab environment.
3. Making provisions to measure the remaining battery charge.
4. Swapping the radio and the power supply components on the board – the radio will be located closer to the edge of the board.
5. Adding a test point for the RTC calibration.
6. Replacing through-hole buttons with surface-mount buttons.

Among the firmware features that I would like to implement in the future are:

1. The ability to resume operation and retrieve data from the Logger after power failure.
2. The mechanism that will keep track of the Logger's temperature and calibrate its ADC when necessary.
3. The radio communication protocol for the wireless data download.
4. For the RTC-based operation, the ability to set different start options: delayed start, start at 00 minutes 00 seconds of the next hour, and start at the specified time.
5. The RTC calibration based on the Logger's temperature.
6. The ability to do measurements when the Logger is connected to the host PC, so the user can verify the Logger's proper operation before deploying it.
7. The ability to display the remaining battery charge.

On the host side, I would like to implement a graphical user interface (GUI) instead of the command-line interface (CLI). As an intermediate step, a wizard, that will help users to set up the Logger, can be implemented.

Additionally, a multi-point calibration as discussed in 5.3 should be performed to provide the required accuracy of temperature measurements when using thermistors.

The final step, after the improved prototype is tested and calibrated, will be to check if it complies with the FCC regulations. Once this requirement is met, the production version of the Logger can be manufactured.

References

[1] T. E. Osterkamp and V. E. Romanovsky, "Evidence for Warming and Thawing of Discontinuous Permafrost in Alaska," *Permafrost and Periglacial Processes,* vol. 10, pp. 17-37, 1999.

[2] T. E. Osterkamp and V. E. Romanovsky, "Characteristics of Changing Permafrost Temperatures in the Alaskan Arctic, U.S.A.," *Arctic and Alpine Research,* vol. 28, no. 3, pp. 267-273, 1996.

[3] Campbell Scientific, Inc., *CR10X Specifications,* Campbell Scientific, Inc., 2006.

[4] Campbell Scientic, Inc., *CR1000 Specifications,* Campbell Scientic, Inc., 2016.

[5] Onset Computer Corp., "HOBO 4-Channel External Data Logger Part # U12-006," Onset Computer Corp., [Online]. Available: http://www.onsetcomp.com/products/data-loggers/u12-006.

[6] Onset Computer Corp., "HOBO 4-Channel External Data Logger Part # U12-008," Onset Computer Corp., [Online]. Available: http://www.onsetcomp.com/products/data-loggers/u12-008.

[7] Onset Computer Corp., "HOBO 4-Channel Analog Data Logger Part # UX120-006M," Onset Computer Corp., [Online]. Available: http://www.onsetcomp.com/products/data-loggers/ux120-006m.

[8] Onset Computer Corp., "HOBO Stainless Temperature (4,900 ft.) Data Logger Part # U12-015," Onset Computer Corp., [Online]. Available: http://www.onsetcomp.com/products/data-loggers/u12-015.

[9] R. O. v. Everdingen, *Multi-Language Glossary of Permafrost and Related Ground-Ice Terms in Chinese, English, French, German, Icelandic, Italian, Norwegian, Polish, Romanian, Russian, Spanish, and Swedish,* Calgary, Alberta, Canada: International Permafrost Association, 2005.

[10] V. Romanovsky, K. Isaksen, D. Drozdov, O. Anisimov, A. Instanes, M. Leibman, D. A. McGuire, N. Shiklomanov, S. Smith and D. Walker, "Changing permafrost and its impacts. In: Snow, Water, Ice and Permafrost in the Arctic (SWIPA) 2017," Arctic Monitoring and Assessment Programme (AMAP), Oslo, Norway, 2017.

[11] Campbell Scientific Inc., "CR1000X Measurement and Control Datalogger," 1 November 2017. [Online]. Available: https://s.campbellsci.com/documents/us/product-brochures/s_cr1000x.pdf.

[12] Accsense, Inc., "VersaLog Model: DCV-2," [Online]. Available: https://caszone-g6x3hkj11rhazj7dhsfo.netdna-ssl.com/wp-content/uploads/2016/09/VersaLog-DCV-2-1.pdf.

[13] Grant Instruments, "Squirrel SQ2040 Wi-Fi high performance data loggers for demanding applications," October 2012. [Online]. Available: http://www.grantinstruments.com/media/50752/sq2040_wi-fi_data_sheet_oct_2012.pdf.

[14] Thermo Fisher Scientific, Inc., "DT85W Series 4 Data Logger," 2017. [Online]. Available: http://www.datataker.com/documents/specifications/1489560581_dataTaker_DT85W.pdf.

[15] Texas Instruments, Inc., "MSP430F665x, MSP430F645x, MSP430F565x, MSP430F535x Mixed Signal Microcontrollers," [Online]. Available: http://www.ti.com/lit/ds/symlink/msp430f5659.pdf.

[16] H. J. R. K. e. a. L. Crovini, "The International Temperature Scale of 1990 (ITS- 90)," *Metrologia,* vol. 77, pp. 3-10, 1990.

[17] H.-G. Schweiger, M. Multerer and H. J. Gores, "Fast Multichannel Precision Thermometer," *IEEE Transactions on Instrumentation and Measurement,* vol. 56, no. 5, pp. 2002-2009, 2007.

[18] F. E. Wudy, D. J. Moosbauer, M. Multerer, G. Schmeer, H.-G. Schweiger, C. Stock, P. F. Hauner, G. A. Suppan and H. J. Gores, "Fast Micro-Kelvin Resolution Thermometer Based on NTC Thermistors," *Journal of Chemical & Engineering Data,* vol. 56, pp. 4823-4828, 2011.

[19] BetaTHERM Sensors, "Series 1 BetaCurve Bare Leaded Thermistors," [Online]. Available: http://www.farnell.com/datasheets/320584.pdf.

[20] R. Burnham and N. Ananthapadamanabhan, "Example Temperature Measurement Applications Using the ADS1247 and ADS1248," Texas Instruments Inc., 2011.

[21] R. Anderson, "Understanding Ratiometric Conversions," Texas Instruments Inc., 2004.

[22] Analog Devices, Inc., Practical Design Techniques for Sensor Signal Conditioning, Prentice Hall, 1999.

[23] Littelfuse, Inc., "PS103J2 - PS Series," [Online]. Available: http://www.littelfuse.com/products/temperature-sensors/leaded-thermistors/interchangeable-thermistors/standard-precision-ps/ps103j2.aspx.

[24] Analog Devices, Inc., "Choosing the Correct Switch, Multiplexer, or Protection Product for Your Application," [Online]. Available: http://www.analog.com/media/en/news-marketing-collateral/product-selection-guide/Choosing_Switches_or_Muxes.pdf.

[25] Texas Instruments, Inc., "ADS1220 Low-Power, Low-Noise, 24-Bit, ADC for Small-Signal Sensors," [Online]. Available: http://www.ti.com/lit/ds/symlink/ads1220.pdf.

[26] Texas Instruments, Inc., "Single-Supply Strain Gauge in a Bridge Configuration Reference Design," [Online]. Available: http://www.ti.com/lit/ug/tidub00/tidub00.pdf.

[27] Texas Instruments, Inc., "ADS1247 24-Bit, 2kSPS, 4-Ch ADC With PGA, Reference, and IDAC for Precision Sensor Measurement," [Online]. Available: http://www.ti.com/lit/ds/symlink/ads1247.pdf.

[28] Texas Instruments, Inc., "3-Wire RTD Measurement System Reference Design, -200°C to 850°C," [Online]. Available: http://www.ti.com/lit/ug/slau520a/slau520a.pdf.

[29] Texas Instruments, Inc., "LM134/LM234/LM334 3-Terminal Adjustable Current Sources," [Online]. Available: http://www.ti.com/lit/ds/symlink/lm134.pdf.

[30] Texas Instruments, Inc., "ADS1148EVM, ADS1248EVM, ADS1148EVM-PDK, and ADS1248EVM-PDK," [Online]. Available: http://www.ti.com/lit/ug/sbau142b/sbau142b.pdf.

[31] Texas Instruments, Inc., "TPS2104, TPS2105 Vaux Power-Distribution Switches.," April 2000. [Online]. Available: http://www.ti.com/lit/ds/symlink/tps2105.pdf.

[32] Texas Instruments, Inc., "TPS780xx 150-mA Low-Dropout Regulator, Ultralow-Power, IQ 500 nA With Pin-Selectable, Dual-Level Output Voltage," [Online]. Available: http://www.ti.com/lit/ds/symlink/tps780.pdf.

[33] Duracell, Inc., "Duracell Ultra 223 Lithium/Manganese Dioxide," [Online]. Available: https://d2ei442zrkqy2u.cloudfront.net/wp-content/uploads/2016/04/Li223_US_OS.pdf.

[34] Texas Instruments, Inc., "TPS22929D Ultra-Small, Low on Resistance Load Switch With Controlled Turn-on," 2014. [Online]. Available: http://www.ti.com/lit/ds/symlink/tps22929d.pdf.

[35] Energizer Brands, LLC, "Product Datasheet Energizer L91 Ultimate Lithium," [Online]. Available: http://data.energizer.com/pdfs/l91.pdf.

[36] Energizer Battery Manufacturing, Inc., "Lithium Iron Disulfide Handbook and Application Manual," 2013.

[37] B. M. Todd, S. Garimella and F. T. Fuller, "A Critical Review of Thermal Issues in Lithium-Ion Batteries," *Journal of the Electrochemical Society,* vol. 158, no. 3, pp. R1-R25, 2011.

[38] Panasonic Industrial Company, "CR-2/3AZ Lithium Manganese Dioxide," 2014. [Online]. Available: https://na.industrial.panasonic.com/sites/default/pidsa/files/clyindricaltypecrseries_datasheets_merged.pdf.

[39] Hitachi Maxell, Ltd., *Lithium Dioxide Manganese Battery,* 2013.

[40] A. M. Aris and B. Shabani, "An experimental study of a lithium ion cell operation at low temperature conditions," in *1st International Conference on Energy and Power, ICEP2016*, Melbourne, Australia, 2016.

[41] Swissbit AG, *Product Data Sheet Industrial MICRO SD Memory Card S-200u Series,* 2016.

[42] Texas Instruments, Inc., "Understanding the Terms and Definitions of LDO Voltage Regulators," 1999.

[43] Texas Instruments, Inc., "TPD2E001 Low-Capacitance 2-Channel ESD-Protection for High-Speed Data Interfaces," March 2016. [Online]. Available: http://www.ti.com/lit/ds/symlink/tpd2e001.pdf.

[44] Anaren Microwave, Inc., "Anaren Integrated Radio A110LR09x User's Manual," 7 October 2015. [Online]. Available: https://cdn.anaren.com/product-documents/AIR/ProprietaryRF/A110LR09X/A110LR09x_Users_Manual.pdf.

[45] H. W. Ott, Electromagnetic Compatibility Engineering, John Wiley & Sons, Inc., 2009.

[46] H. W. Johnson and M. Graham, High-Speed Digital Design A Handbook of Black Magic, Englewood Cliffs, NJ: Prentice Hall PTR, 1993.

[47] E. Bogatin, Signal and Power Integrity - Simplified. Second Edition, Prentice Hall, 2009.

[48] C. Speck and S. Schauer, "MMC Lib Version 1.1," Texas Instruments, Inc., 2005.

[49] J. S. Steinhart and S. R. Hart, "Calibration curves for thermistors," *Deep-Sea Research,* vol. 15, pp. 497-503, 1968.

[50] C. B. Moler, Numerical Computing with MATLAB, Philadelphia, Pennsylvania: Society for Industrial and Applied Mathematics, 2004.

[51] Campbell Scientific, Inc., "CR1000 Measurement and Control Datalogger," [Online]. Available: https://www.campbellsci.com/cr1000.

[52] Analog Devices, Inc., "CMOS, +1.8 V to +5.5 V/±2.5 V, 2.5 Ohm Low-Voltage, 8-/16-Channel Multiplexers ADG706/ADG707," 2016. [Online]. Available: http://www.analog.com/media/en/technical-documentation/data-sheets/ADG706_707.pdf.

[53] J. Wise, NIST Measurement Services: Liquid-In_Glass Thermometer Calibration Service, Washington: U.S. Government Printing Office, 1988.

[54] Fluke Corporation, "7008/ 7040/ 7037/ 7012/ 7011 Refrigerated Temperature Calibration Baths," Fluke Corporation, [Online]. Available: https://us.flukecal.com/products/temperature-

calibration/calibration-baths/standard-calibration-baths/7008-7040-7037-7012-70?quicktabs_product_details=2. [Accessed 25 10 2018].

[55] K. G. Hubbard, X. Lin and E. A. Walter-Shea, "The Effectiveness of the ASOS, MMTS, Gill, and CRS Air Temperature Radiation Shields," *Journal of Applied Meteorology,* 1965.

[56] Keysight Technologies, "Keysight Model 66319B/D, 66321B/D Mobile Communications DC Source User's Guide," 2014.

[57] Texas Instruments Inc., "MSP430x5xx and MSP430x6xx Family User's Guide," June 2008–Revised May 2015. [Online]. Available: http://www.ti.com/lit/ug/slau208p/slau208p.pdf.

[58] Texas Instruments, Inc., "TLV2450, TLV2451,TLV2452, TLV2453,TLV2454, TLV2455, TLV245xA Family of 23-uA 220-KHz Rail-to-Rail Input/Output Operational Amplifiers With Shutdown," January 2005. [Online]. Available: http://www.ti.com/lit/ds/symlink/tlv2454.pdf.

[59] L. A. Williams and B. A. Wooley, "A Third-Order Sigma-Delta Modulator with Extended Dynamic Range," *IEEE Journal of Solid-State Circuits,* vol. 29, no. 3, pp. 193-194, 1994.

Appendix A

The Logger's Schematic. VE Scheme Configuration #2.

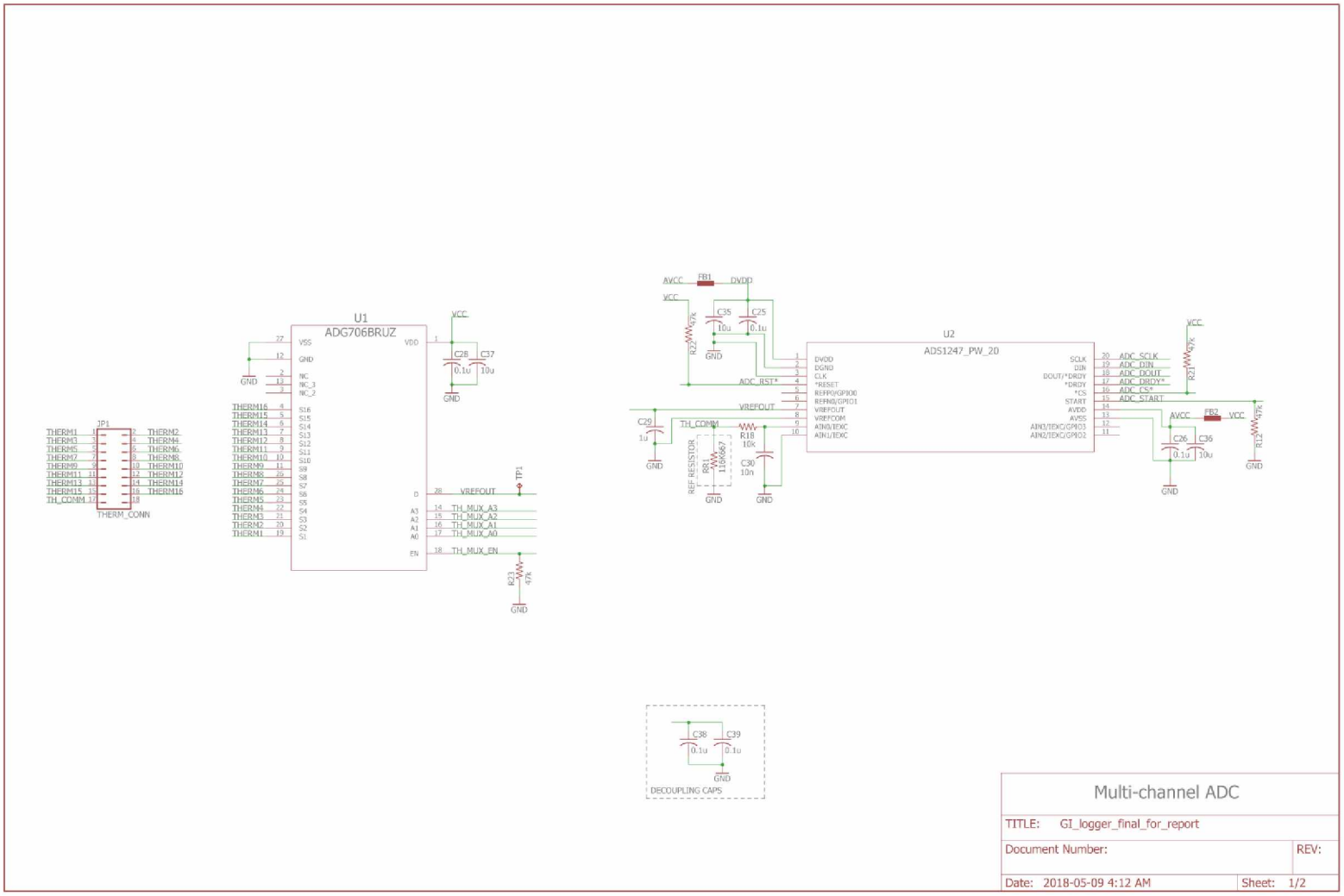

Figure A-1. The Logger's schematic sheet 1

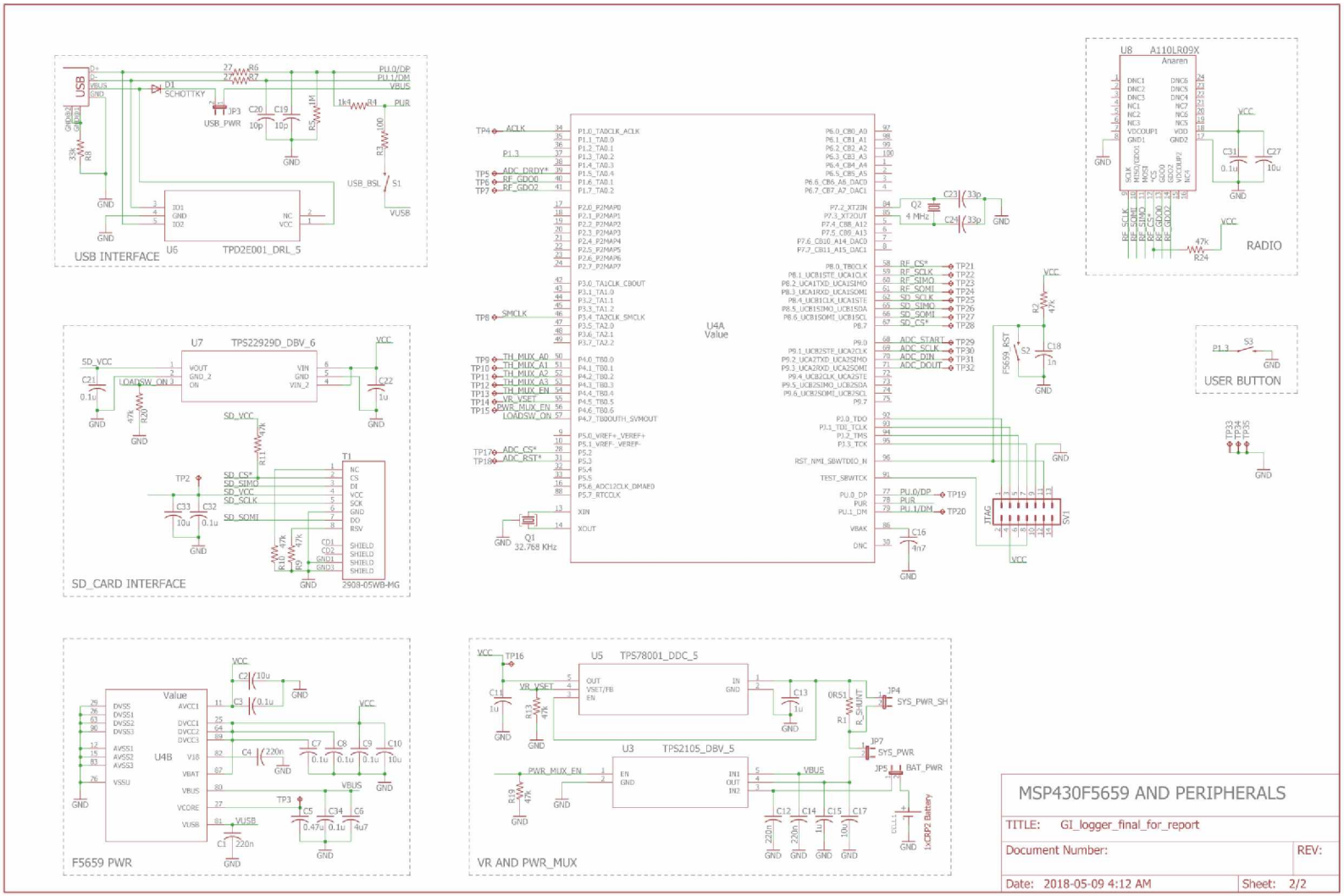

Figure A-2. The Logger's schematic sheet 2

Appendix B

The Logger's Printed Circuit Board

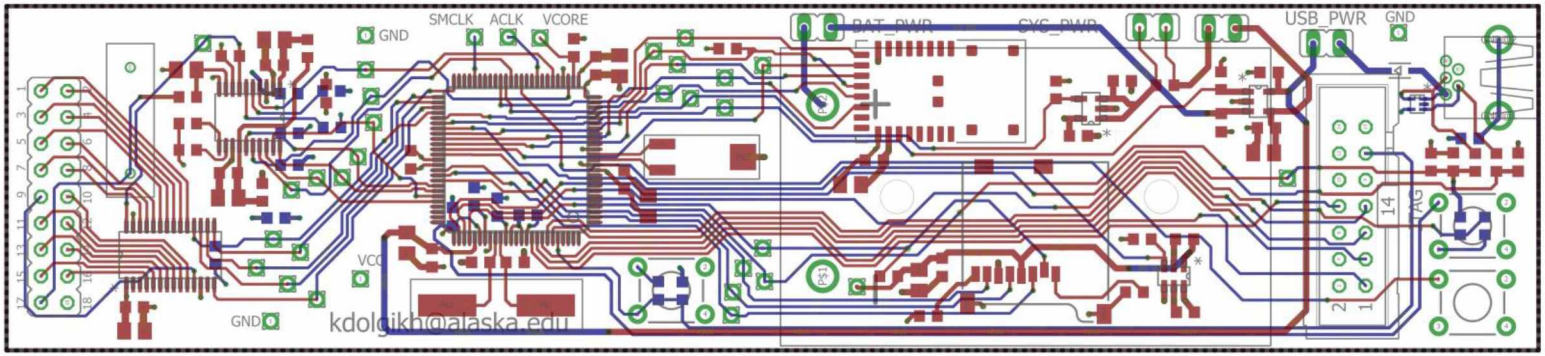

Figure B-1. Top traces (red), bottom traces (blue), top/bottom silk screen(grey), pads/vias (green)

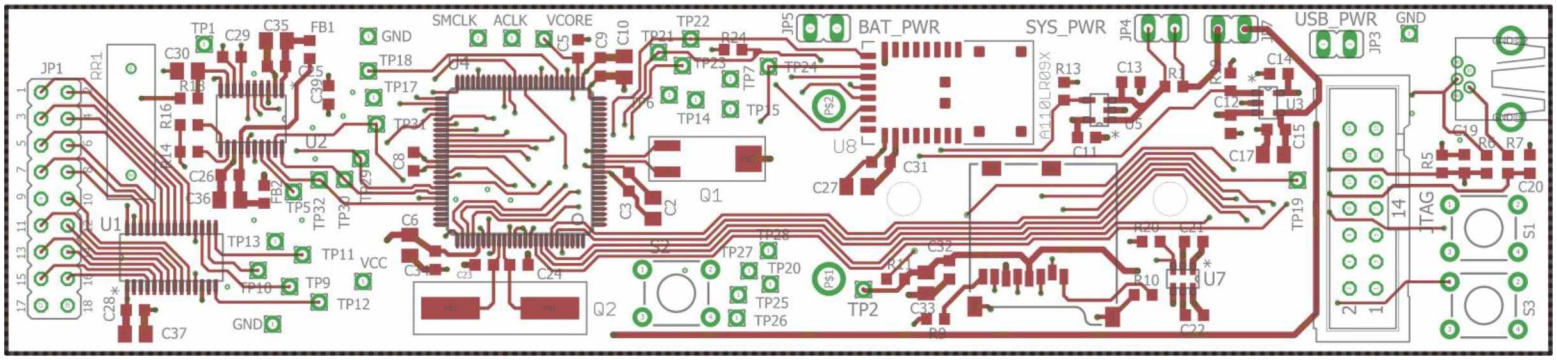

Figure B-2. Top traces (red), top silk screen and names (grey), pads/vias (green)

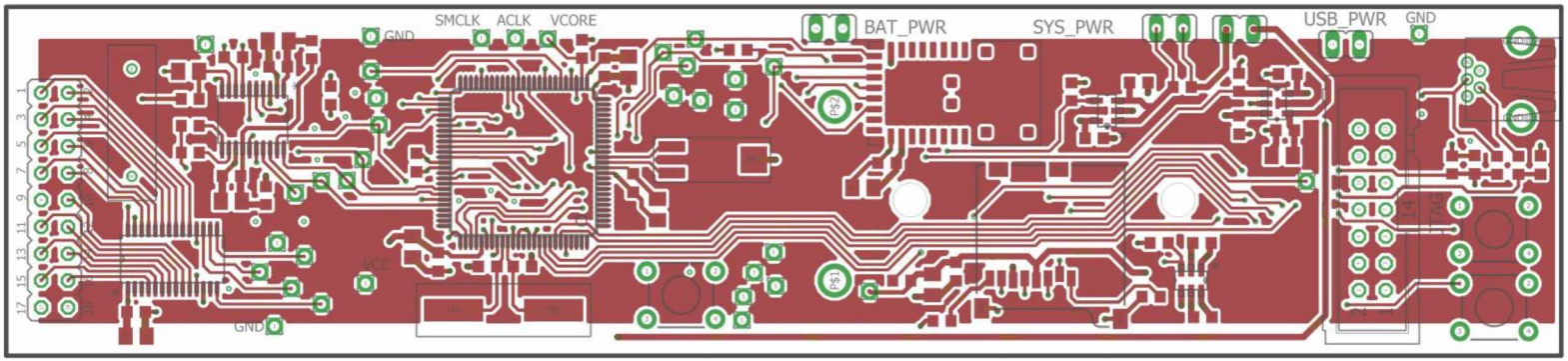

Figure B-3. Top layer (layer 1)

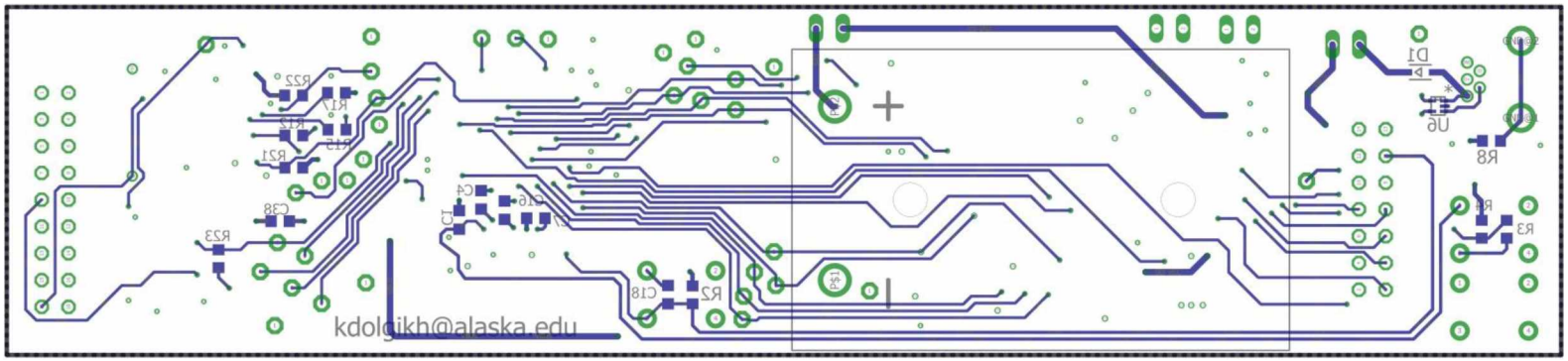

Figure B-4. Bottom traces (blue), bottom silk screen and names (grey), pads/vias (green)

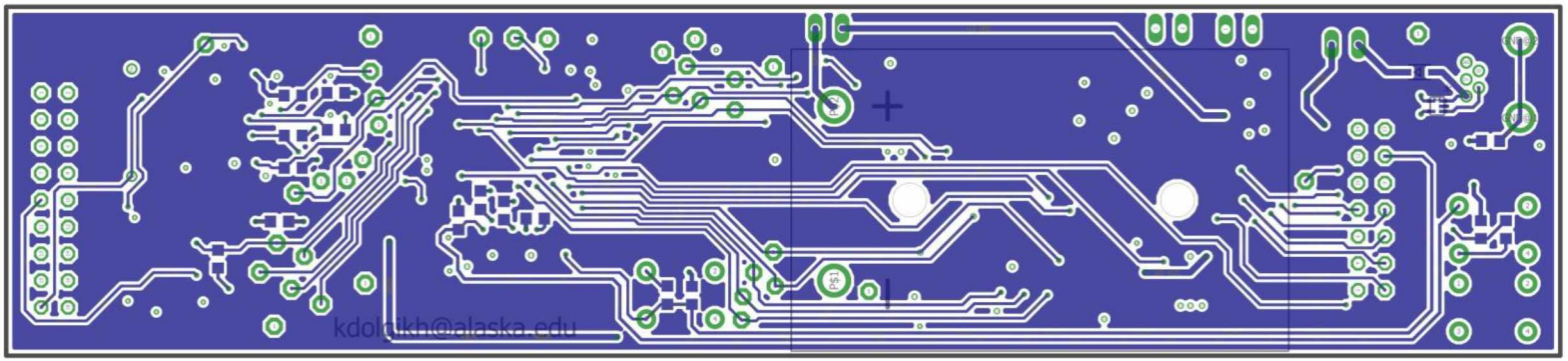

Figure B-5. Bottom layer (layer 4)

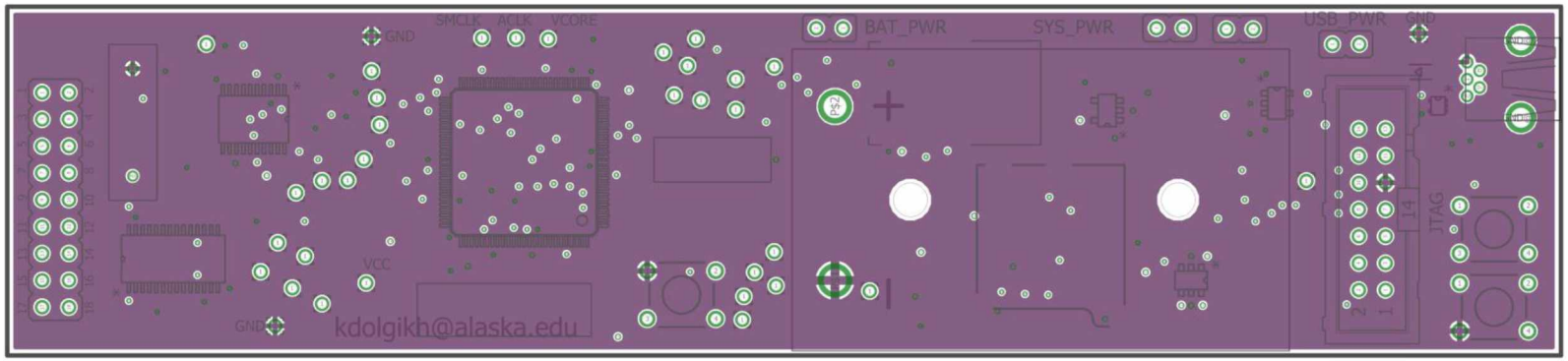

Figure B- 6. Ground plane (layer 2)

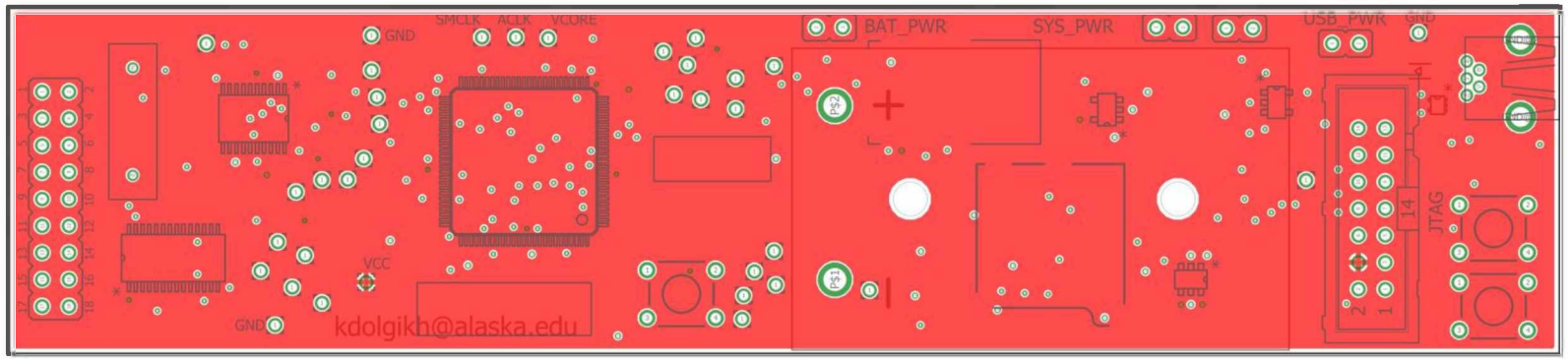

Figure B-7. Power plane (layer 3)

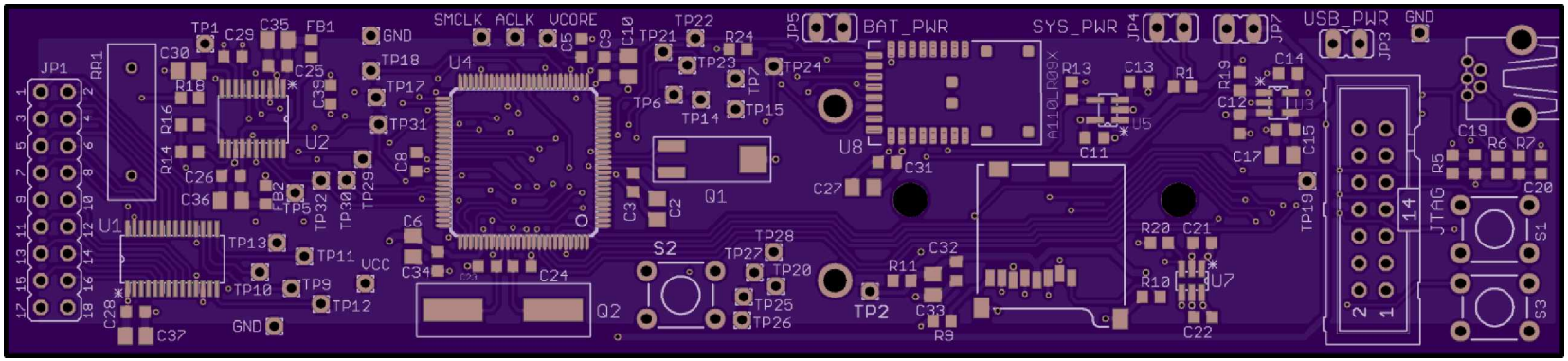

Figure B-8. Manufactured board's top

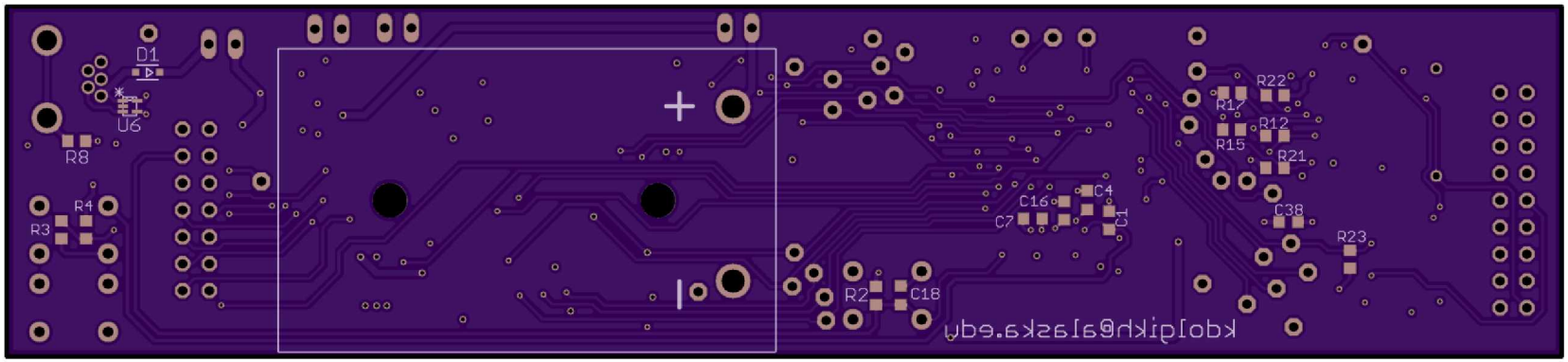

Figure B-9. Manufactured board's bottom

Appendix C

The Test Bed with Resistors

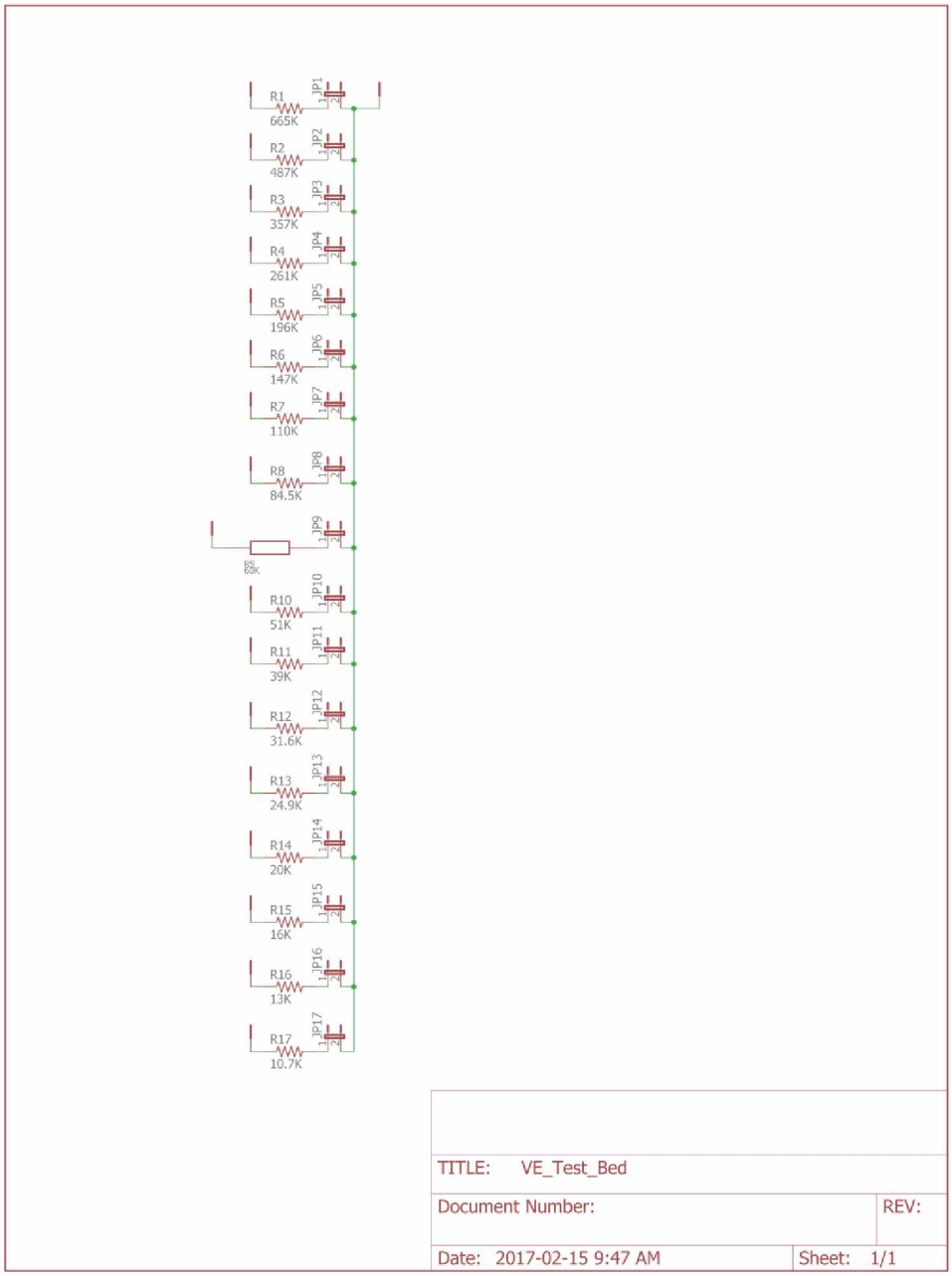

Figure C-1. The test bed's schematic

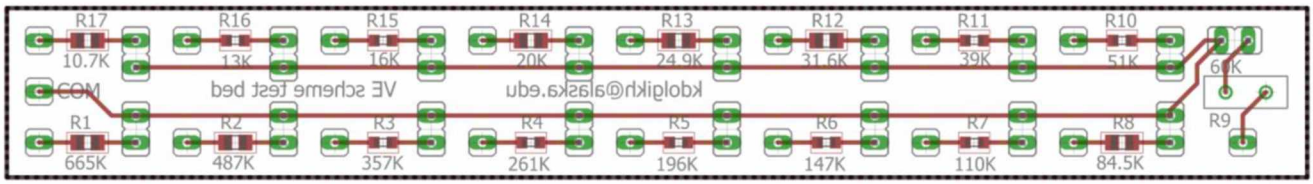

Figure C-2. The test bed's PCB layout

Appendix D

The ADS1247 PGA Simulation

As discussed in 5.1, the accuracy of the Logger was determined in the experiment, in which the Logger measured 18 resistors representing the thermistor at different ambient temperatures. It has been noted that the measurement accuracy degrades as the resistor value becomes larger (5.2.1.1). To understand the reasons behind this behavior, the circuit representing the ADS1247 PGA with the connected sensing circuit was simulated and studied in PSPICE. The simulation results are presented in this Appendix.

The simplified diagram of the ADS1247 PGA is shown in Figure D-1. Values of unlabeled variable resistors and a single fixed resistor, which determine the PGA's gain, are not present. In the diagram, resistors of the ADC's input low-pass filter are labeled as "R". A1 and A2 are op-amps. The fact that the input current of the PGA changes its sign, as indicated in Table 7 in [15], suggests that rail-to-rail op-amps are used in the PGA.

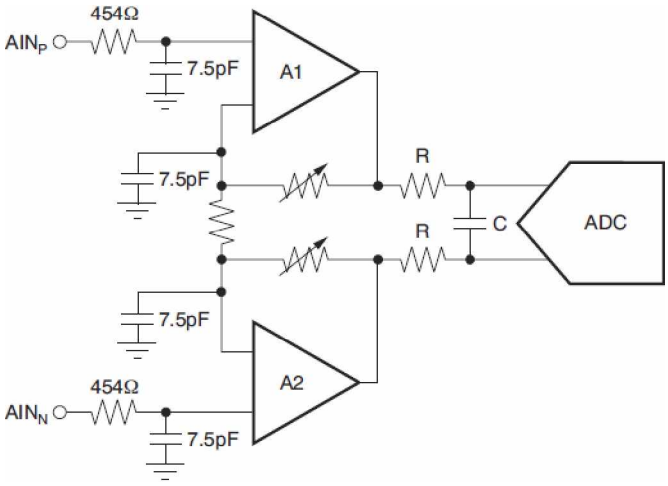

Figure D-1. The simplified diagram of the ADS1247 PGA taken from [15]

The voltage gain of this configuration can be calculated using Eq. D.1 below:

$$Gain = 1 + \frac{2 * R_{VAR}}{R_{FIXED}} \qquad \text{D.1}$$

where R_{VAR} is the variable resistors' value, R_{FIXED} is the fixed resistor's value. As described in 2.1.5, the PGA gain setting of 1 is used in the Logger. For unity gain, both variable resistors should be set to zero (short-circuited). In this case, gain is independent from the fixed resistor's value.

The simulated circuit is presented in Figure D-2. A Texas Instruments TLV2451 op-amp model is used because its electrical characteristics are close to the characteristics of the ADC's PGA: it is a single supply rail-to rail input/output op-amp with a bias current of 0.9 nA, an input offset voltage of 300 µV, and a differential input resistance of 1 GΩ [58].

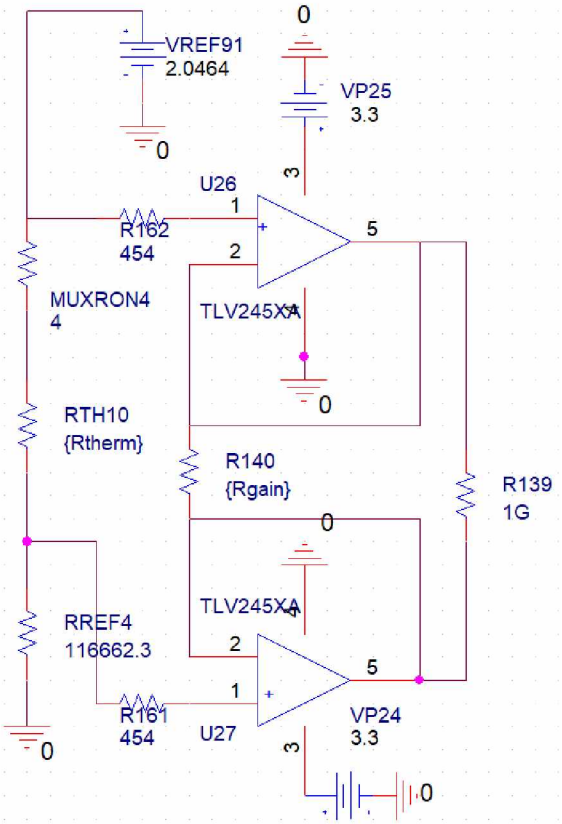

Figure D-2. The circuit simulating the ADS1247 PGA

Values of the reference voltage "VREF91" of 2.0464 V and the reference resistor "RREF4" of 116662.3 Ω were measured on Board 2. The multiplexer on resistance "MUXRON4" of 4 Ω was determined using figure 1 in [52]. In the simulation, the thermistor resistance "RTH10" (R_{therm}) was varied linearly from 1 kΩ to 700 kΩ.

In ADS1247, the PGA output is connected to the 3rd order sigma-delta modulator. A widely adopted 3-rd order modulator architecture is based on the work described in [59]. In this architecture, the modulator input stage consists of the switching-capacitor integrator, which has high input resistance. Assuming that the ADS1247 has the same modulator architecture as described in [59], the PGA load resistor "R139" was set to a high value of 1 GΩ.

The value of the fixed resistor "R140" (R_{GAIN}) was set to 100 kΩ. Simulation where R_{GAIN} was set to 10kΩ (top trace), 50kΩ, 100kΩ, and 200kΩ (bottom trace) showed no significant variation in the circuit gain except for the case of 10 kΩ (Figure D-3). The PGA's input resistance is influenced by the value of R_{GAIN} as well. For R_{GAIN} values of 50 kΩ, 100 kΩ, and 200 kΩ, the difference in the PGA input resistances was around 500 Ω, which is negligible compared to the value of the input resistance itself, which varies from 31.5 MΩ at $R_{therm} = 1$ kΩ to 3.2 GΩ at $R_{therm} = 700$ kΩ (Figure D-4). The input resistance shown in Figure D-4 is obtained by dividing the voltage across the PGA to the current flowing into the non-inverting terminal of the PGA's top op-amp.

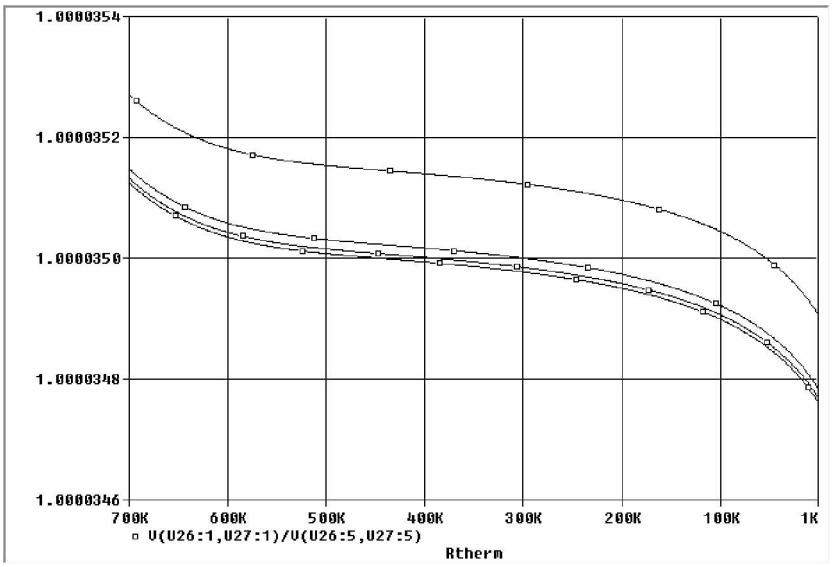

Figure D-3. Effect of the fixed resistor value on the PGA gain

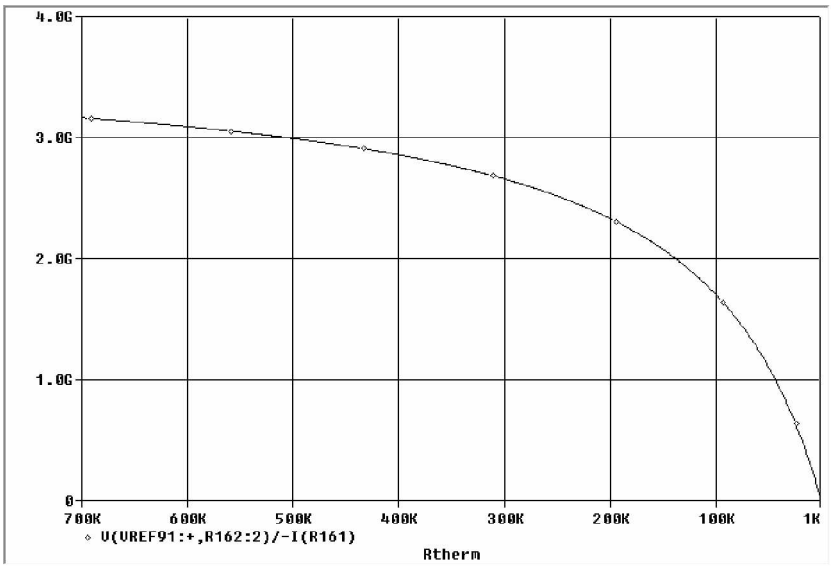

Figure D-4. The PGA input resistance

As discussed in 2.1.4, equation 2.4, which is used to calculate resistance measured by the ADS1247, is derived from the voltage-divider circuit of the configuration #3 (2.1.2). The configuration #3 reproduced for this simulation is shown in Figure D-5.

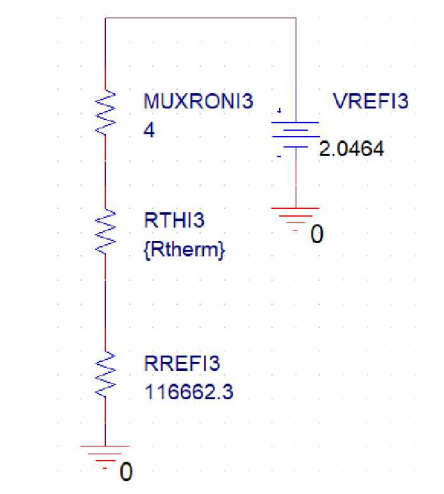

Figure D-5. The configuration #3 circuit used in simulation

In order to verify that the simulation produces the result similar to the real-life experiment, it is necessary to solve the reverse problem of finding the measured resistance for the given $code$ value. To calculate the $code$ from equation 2.4, equation 2.3 can be used, where simulated input voltage of the PGA is substituted as V_{MEAS}. In this case it is safer to use the PGA's input voltage than the output voltage because the output stage components values are not known. On the other hand, the influence of these components on the input stage is minimal. The input stage parameters are mostly defined by the op-amp parameters and the sensing circuit configuration. Therefore, it is assumed that the ADS1247 PGA has the gain of 1, so the PGA's output voltage is the same as its input voltage.

The difference between the measured resistance calculated with equations 2.3 and 2.4 using the PGA input voltage as V_{MEAS} and the resistance $(R_{therm} + R_{ON})$ is shown in Figure D-6. Because the PGA's resistance is connected in parallel to the thermistor and the multiplexer ON resistances, the effective resistance of this circuit is lower than the resistance of the thermistor and the multiplexer ON resistances alone. This causes the PGA's input voltage, and hence the output voltage, to be higher than the voltage across the thermistor and the multiplexer ON resistances in the VE configuration #3 (Figure D-5). Inserting this higher voltage into Eq. 2.4 yields the larger value of the calculated measured resistance, as illustrated in Figure D-6.

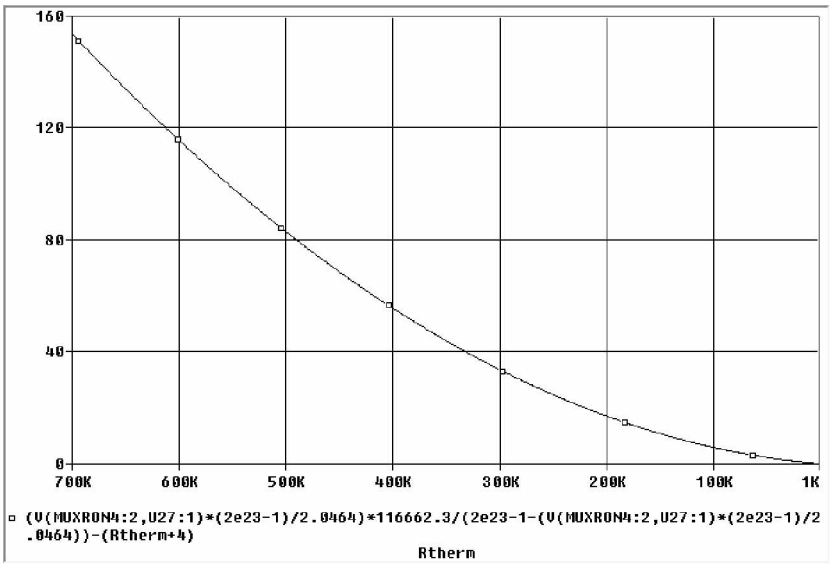

Figure D-6. Calculating the measured resistance using the PGA input voltage

The initial naïve approach to compensate for the PGA's input resistance was to treat the measured resistance as the thermistor and multiplexer resistances with the parallelly connected PGA input resistance, which was assumed to have some definite value. However, applying this approach to the measured resistance only increased the error: Figure D-7 illustrates this effect.

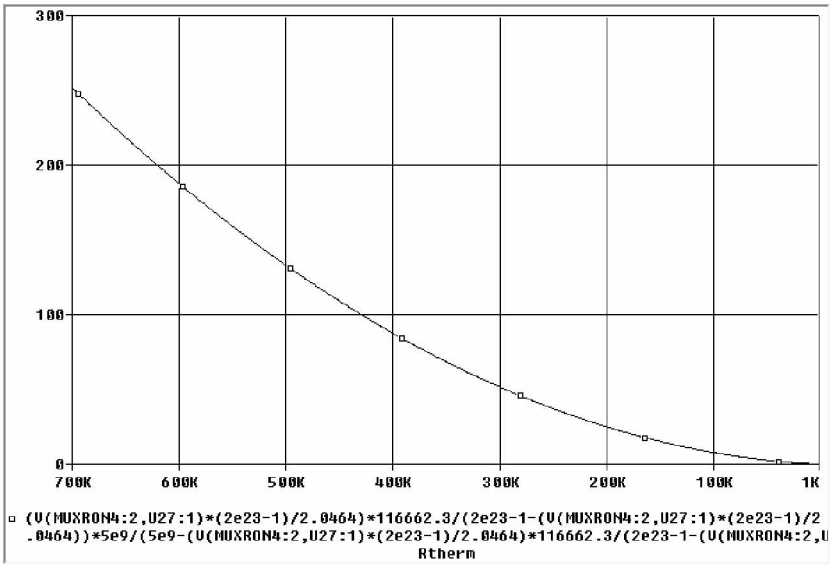

Figure D-7. The erroneous approach to calculate the measured resistance increased the error

Comparing Figure 5-7 from 5.2.1.1 and Figure D-7, it is seen that the simulated results are five times smaller than the experimental results. This may be caused by several reasons, but the biggest contribution comes from the op-amp model that is used in simulation.

Apparently, the issue with accuracy, which degrades as the thermistor resistance increases, needed a different solution. Further research on the topic revealed that the PGA input resistance cannot be represented by a single resistor. At the minimum, each op-amp has a differential and two common mode input resistances. Additionally, the circuit configuration influences the PGA input resistance. In general, it is difficult to derive the analytical equation for the PGA input resistance, which can be used to improve the accuracy of the measurements. For these reasons, calibration seemed an easier yet as efficient way to overcome this issue, which is described in 5.2.1.1.

Appendix E

The Experiment's Data for the Logger

Table E-1. Statistics for measured voltages for Board 2, DR = 10 SPS, 5000 samples

Resistor ID	Measured voltage				
	Min deviation, μV	Average, V	Max deviation, μV	Range (peak-to-peak noise voltage), μV	Standard deviation, μV
665K	-16.4262	1.74162	18.9465	35.3727	5.0884
487K	-17.7638	1.65165	18.8287	36.5925	5.2314
357K	-21.1399	1.54255	22.7711	43.9110	7.4049
261K	-16.5013	1.41476	18.3836	34.8848	5.5268
196K	-17.5703	1.28343	19.0222	36.5925	6.1365
147K	-17.4812	1.14141	20.0871	37.5683	5.8883
110K	-16.6404	0.99254	16.0489	32.6893	4.8641
85K	-16.9916	0.85998	20.0888	37.0804	5.0781
60K	-15.4352	0.69525	17.2541	32.6893	4.7017
51K	-19.6634	0.62268	16.1973	35.8606	4.5661
39K	-15.6605	0.51289	13.8574	29.5179	4.6994
32K	-16.6738	0.43629	15.5276	32.2014	4.5599
25K	-15.5476	0.36002	15.6780	31.2256	4.3915
20K	-15.989	0.29958	17.1882	33.1772	4.3989
16K	-15.4091	0.24695	15.8164	31.2256	4.4492
13K	-16.0823	0.20529	17.8268	33.909	4.2296
11K	-15.2113	0.17205	14.7945	30.0058	4.4292
5K	-15.9917	0.083902	15.2338	31.2256	4.3939

Table E-2. Statistics for measured voltages for Board 1, DR = 10 SPS, 510 samples

Resistor ID	Measured voltage				
	Min deviation, μV	Average, V	Max deviation, μV	Range (peak-to-peak noise voltage), μV	Standard deviation, μV
665K	-18.3149	1.74159	18.5215	36.8364	6.4364
487K	-16.6632	1.65161	16.5140	33.1772	6.3372
357K	-16.7484	1.54249	16.6728	33.4211	5.6969
261K	-13.5894	1.41470	13.0011	26.5905	4.0860
196K	-16.1471	1.28336	15.8103	31.9574	5.6020
147K	-14.2266	1.14135	15.5353	29.7619	5.2601
110K	-16.2882	0.99246	22.2559	38.5441	6.4955
85K	-14.8824	0.85991	12.1961	27.0784	4.9750
60K	-21.1854	0.69519	19.7982	40.9836	7.5876
51K	-15.6525	0.62261	15.3291	30.9816	5.6150
39K	-17.1081	0.51284	12.6538	29.7619	5.0863
32K	-13.8994	0.43623	14.3988	28.2982	5.2711
25K	-15.9931	0.35997	15.9644	31.9574	5.025
20K	-11.3068	0.29954	12.8442	24.1510	4.5544
16K	-19.1802	0.24690	15.2167	34.3969	4.7401
13K	-11.3327	0.20523	13.7942	25.1268	3.9548
11K	-13.5014	0.17200	14.5528	28.0542	4.0995
5K	-10.0809	0.08383	11.6307	21.7115	3.8273

Table E-3. Statistics for measured voltages for Board 1, DR = 5 SPS, 510 samples

Resistor ID	Measured voltage				
	Min deviation, µV	Average, V	Max deviation, µV	Range (peak-to-peak noise voltage), µV	Standard deviation, µV
665K	-6.3963	1.74157	8.2407	14.6370	2.6679
487K	-8.2029	1.65159	8.8736	17.0765	2.9290
357K	-8.1575	1.54249	7.9432	16.1007	3.0496
261K	-7.4878	1.41470	8.6129	16.1007	3.0774
196K	-10.8864	1.28335	8.3857	19.272	3.7441
147K	-8.3158	1.14133	9.0046	17.3204	2.9186
110K	-7.6820	0.99246	7.1989	14.8809	2.4962
85K	-7.7595	0.85990	8.3412	16.1007	3.2122
60K	-6.9521	0.69516	8.1728	15.1249	2.6168
51K	-6.9880	0.62259	8.1369	15.1249	2.7921
39K	-7.5543	0.51282	7.8145	15.3688	2.7751
32K	-8.0312	0.43621	7.8255	15.8567	2.9301
25K	8.7865	0.35995	6.3384	15.1249	2.7079
20K	-7.021	0.29951	7.3721	14.393	2.7408
16K	-8.8908	0.24688	8.6736	17.5644	2.9519
13K	-7.9049	0.20522	9.1716	17.0765	2.6976
11K	-7.9662	0.17198	7.6466	15.6128	2.7941
5K	-7.0454	0.08383	7.3477	14.3930	2.5780

Table E-4. Uncalibrated equivalent temperature accuracy of Board 2 at DR = 10 SPS, 5000 samples

Resistor ID	10 kΩ thermistor			20 kΩ thermistor		
	Equivalent temperature accuracy, °C			Equivalent temperature accuracy, °C		
	Min deviation	Average	Max deviation	Min deviation	Average	Max deviation
665K	-	-	-	0.0282581	0.0297801	0.0312598
487K	0.0212158	0.0220226	0.022877737	0.0228141	0.0236817	0.0246013
357K	0.0171478	0.0181815	0.01918305	0.0184642	0.0195773	0.0206557
261K	0.0149327	0.0155165	0.016166758	0.0161012	0.0167306	0.0174318
196K	0.0130419	0.0137160	0.014349376	0.0140819	0.0148097	0.0154936
147K	0.0112865	0.0118558	0.012509976	0.0122037	0.0128192	0.0135265
110K	0.0102977	0.0108505	0.011383725	0.0111509	0.0117496	0.0123269
85K	0.0083754	0.0092455	0.010326815	0.0090821	0.0100256	0.0111982
60K	0.0068851	0.0074971	0.008181093	0.0074861	0.0081514	0.0088952
51K	0.0068486	0.0076906	0.008384107	0.0074565	0.0083732	0.0091282
39K	0.0055047	0.0062851	0.006975669	0.0060045	0.0068557	0.0076089
32K	0.0053994	0.0063544	0.007243642	0.0058946	0.0069370	0.0079078
25K	0.0053308	0.0063924	0.00746284	0.0058289	0.0069895	0.0081599
20K	0.0050201	0.0063224	0.007722195	0.0054977	0.0069240	0.0084571
16K	0.0045434	0.0060646	0.007626023	0.0049827	0.0066511	0.0083634
13K	0.0039999	0.0059181	0.008044088	0.0043902	0.0064957	0.0088295
11K	0.0041777	0.0063597	0.008481623	0.0045950	0.0069951	0.0093293
5K	0.0014927	0.0064920	0.011242917	-	-	-

Table E-5. Uncalibrated equivalent temperature accuracy of Board 1 at DR = 10 SPS, 510 samples

Resistor ID	10 kΩ thermistor			20 kΩ thermistor		
	Equivalent temperature accuracy, °C			Equivalent temperature accuracy, °C		
	Min deviation	Average	Max deviation	Min deviation	Average	Max deviation
665K	-	-	-	0.0273573	0.0284203	0.0294952
487K	0.0194620	0.0202187	0.0209687	0.0209282	0.0217420	0.0225484
357K	0.0152992	0.0159583	0.0166144	0.0164737	0.0171833	0.0178898
261K	0.0130637	0.0135444	0.0140042	0.0140859	0.0146042	0.0151001
196K	0.0112552	0.0117927	0.0123190	0.0121527	0.0127331	0.0133013
147K	0.0101128	0.0105761	0.0110820	0.0109345	0.0114355	0.0119825
110K	0.0082658	0.0088069	0.0095463	0.0089506	0.0095366	0.0103372
85K	0.0070974	0.0076195	0.0080474	0.0076963	0.0082625	0.0087265
60K	0.0056316	0.0064715	0.0072564	0.0061232	0.0070364	0.0078897
51K	0.0056079	0.0062781	0.0069345	0.0061057	0.0068354	0.0075500
39K	0.0049154	0.0057679	0.0063984	0.0053616	0.0062915	0.0069793
32K	0.0048973	0.0056933	0.0065179	0.0053464	0.0062154	0.0071156
25K	0.0052197	0.0063116	0.0074016	0.0057073	0.0069012	0.0080929
20K	0.0057866	0.0067075	0.0077534	0.0063372	0.0073457	0.0084913
16K	0.0045289	0.0064224	0.0079244	0.0049669	0.0070434	0.0086907
13K	0.0032672	0.0046179	0.0062640	0.0035859	0.0050685	0.0068754
11K	0.0061127	0.0080491	0.0101361	0.0067235	0.0088535	0.0111492
5K	0.0007714	0.0039194	0.0075509	-	-	-

Table E-6. Uncalibrated equivalent temperature accuracy of Board 1 at DR = 5 SPS, 510 samples

Resistor ID	10 kΩ thermistor			20 kΩ thermistor		
	Equivalent temperature accuracy, °C			Equivalent temperature accuracy, °C		
	Min deviation	Average	Max deviation	Min deviation	Average	Max deviation
665K	-	-	-	0.0269751	0.0273463	0.0278245
487K	0.0193069	0.0196794	0.0200824	0.0207614	0.0211620	0.0215954
357K	0.0154528	0.0157738	0.0160864	0.0166391	0.0169847	0.0173213
261K	0.0131586	0.0134235	0.0137281	0.0141883	0.0144739	0.0148024
196K	0.0111740	0.0115364	0.0118155	0.0120650	0.0124563	0.0127577
147K	0.0096759	0.0099467	0.0102399	0.0104621	0.0107549	0.0110720
110K	0.0085413	0.0087965	0.0090357	0.0092490	0.0095254	0.0097843
85K	0.0069005	0.0071728	0.0074654	0.0074828	0.0077780	0.0080954
60K	0.0051867	0.0054623	0.0057864	0.0056395	0.0059391	0.0062914
51K	0.0051169	0.0054162	0.0057646	0.0055712	0.0058969	0.0062763
39K	0.0044534	0.0048299	0.0052193	0.0048577	0.0052683	0.0056931
32K	0.0040590	0.0045190	0.0049672	0.0044312	0.0049334	0.0054227
25K	0.0042370	0.0048369	0.0052697	0.0046329	0.0052888	0.0057620
20K	0.0042369	0.0048087	0.0054092	0.0046400	0.0052662	0.0059238
16K	0.0033488	0.0042266	0.0050828	0.0036727	0.0046353	0.0055744
13K	0.0032672	0.0042100	0.0053039	0.0035859	0.0046208	0.0058215
11K	0.0031733	0.0043161	0.0054129	0.0034903	0.0047472	0.0059537
5K	0.0022189	0.0044189	0.0067131	-	-	-

Appendix F

The Microcontroller's Firmware

main.c

```
/* --COPYRIGHT--,BSD
 * Copyright (c) 2016, Texas Instruments Incorporated
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * *  Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * *  Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * *  Neither the name of Texas Instruments Incorporated nor the names of
 *    its contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
 * OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 * --/COPYRIGHT--*/

/*
 * Uses TI's USB library
 */

#include "driverlib.h"
#include "USB_config/descriptors.h"
#include "USB_API/USB_Common/device.h"
#include "USB_API/USB_Common/usb.h"
#include "USB_API/USB_CDC_API/UsbCdc.h"
#include "USB_app/usbConstructs.h"
#include "mux_adg706_driver.h"
#include "adc_driver.h"
#include "states.h"
#include "init.h"
#include "send_over_usb.h"
#include "sleep_timer.h"
#include "sdcard.h"
#include "measure.h"
#include "real_time_clock.h"
```

```
device_state dev_state = adc_measuring;          // Device state
operation_mode op_mode = rtc_operation;
uint32_t max_segments_sd = 0;

void main (void)
{
    device_init();                               // Init device modules
    max_segments_sd = num_segments_sdc(sdcSize); // Determine max num of segments on SD-card
    __enable_interrupt();                        // Enable interrupts globally
    Radio_Init();
    USB_setup(FALSE, TRUE);                      // Init USB & events

    while (1)
    {
        switch (USB_getConnectionState())
        {
            case ST_ENUM_ACTIVE: //4
                if (USB_DataReceived_event == TRUE)
              {
                      USB_DataReceived_event = FALSE;
                      usb_command_line();
              }
                if(USBconnected)
                      LPM0;   // wait until data is received
              break;

            // USB disconnected
            case ST_PHYS_DISCONNECTED: //1
              rt1ps_count=0;
              if(start_immediately) // Start only when user commands it
              {
                      RTCPS1CTL &= ~RT1PSIFG; // clear interrupt flag
                      RTCPS1CTL |= RT1PSIE;   // En RT1PS interrupt to sync the measurement
              }
              LPM3; // Wait until RT1PSIFG is hit or sleep here until USB wakes the device

              if (dev_state == adc_measuring)
              {
                      if (!CalEn)
                      {
                              adc_calibration(adc_data_rate);
                              adc_setup(adc_data_rate, adc_mux1_vrefcon);
                      }
                      switch(op_mode)
                      {
                              case bulk_operation:
                                      dev_state =
measure_bulk_all_channels_store_flash(num_segm_bulk);
                              break;
                              case rtc_operation:
                                      dev_state = measure_all_channels_store_flash_sdc();
                                      break;
                      }
              }
              if (dev_state == adc_rst)
                      dev_state = adc_reset();
              if (dev_state == usb_connected)
                      usb_connect2host();
              if (dev_state == parameter_error)
                      _NOP();
```

```
            if (dev_state == dev_mem_full)
                    _NOP();
            if (dev_state == dev_malfunction)
                    _NOP();
            break;

          // These cases are executed while the device is enumerated but suspended
          // by the host, or connected to a powered hub without a USB host
          // present.
          case ST_PHYS_CONNECTED_NOENUM: //2
          case ST_ENUM_SUSPENDED: //5
          case ST_PHYS_CONNECTED_NOENUM_SUSP: //6
            LPM3;
             break;

          // The default is executed for the momentary state
          // ST_ENUM_IN_PROGRESS.  Usually, this state only last a few
          // seconds.  Be sure not to enter LPM3 in this state; USB
          // communication is taking place here, and therefore the mode must
          // be LPM0 or active-CPU.
          case ST_ENUM_IN_PROGRESS: //3
          default:;
        }
    }  //while(1)
} //main()

#pragma vector=RTC_VECTOR
__interrupt void RTC_B_ISR (void) {
    switch (RTCIV) {
        case (RTCIV_RTCRDYIFG):
                    RTCCTL0 &= ~RTCRDYIE; // Disable RTC Ready interrupt
            LPM0_EXIT;
            break;
        case(RTCIV_RTCAIFG):
              break;
        case(RTCIV_RT0PSIFG):
              break;
        case(RTCIV_RT1PSIFG):
              if (start_immediately) // Sync measurement with 2 sec boundary
              {
                    start_immediately = FALSE;
                    LPM3_EXIT; // Used only once
              }
              else
              {   // Count interrupts to calculate meas rate
                            rt1ps_count++; // volatile
                            if (rt1ps_count == (meas_rate >> 1)) // Divide by 2
                            {
                                  rt1ps_count = 0;
                                  LPM3_EXIT;
                            }
              }
              break;
        case(RTCIV_RTCOFIFG):
              // Add procedure to recover from oscillator fault
              break;
        default:
            break;
    }
}
```

```
#pragma vector=PORT1_VECTOR
__interrupt void PORT1_ISR (void) {
	switch (P1IV) {
			case (P1IV_P1IFG3):   // user button
				LPM3_EXIT;
				break;
			case (P1IV_P1IFG5):
				switch(dev_state)
				{
					case(adc_set):
					case(adc_cal):
					case(adc_measuring):
						LPM3_EXIT;
						break;
					case(usb_enumerated):
						LPM0_EXIT;
					default:
						break;
				}
				break;
			default:
				break;
			}
}

#pragma vector=TIMER2_A1_VECTOR
__interrupt void Timer2_A1_ISR (void) {
	switch (TA2IV) {
		case (TA2IV_TACCR1):
			timer_event = TRUE; // volatile
			LPM3_EXIT;
			break;
		case (TA2IV_TAIFG):
			break;
		default:
			break;
		}
}

#if defined(__TI_COMPILER_VERSION__) || (__IAR_SYSTEMS_ICC__)
#pragma vector = UNMI_VECTOR
__interrupt void UNMI_ISR (void)
#elif defined(__GNUC__) && (__MSP430__)
void __attribute__ ((interrupt(UNMI_VECTOR))) UNMI_ISR (void)
#else
#error Compiler not found!
#endif
{
    switch (__even_in_range(SYSUNIV, SYSUNIV_BUSIFG ))
    {
        case SYSUNIV_NONE:
            __no_operation();
            break;
        case SYSUNIV_NMIIFG:
            __no_operation();
            break;
        case SYSUNIV_OFIFG:
            UCS_clearFaultFlag(UCS_XT2OFFG);
            UCS_clearFaultFlag(UCS_DCOFFG);
            SFR_clearInterrupt(SFR_OSCILLATOR_FAULT_INTERRUPT);
```

```
            break;
        case SYSUNIV_ACCVIFG:
            __no_operation();
            break;
        case SYSUNIV_BUSIFG:
            // If the CPU accesses USB memory while the USB module is
            // suspended, a "bus error" can occur.  This generates an NMI.  If
            // USB is automatically disconnecting in your software, set a
            // breakpoint here and see if execution hits it.  See the
            // Programmer's Guide for more information.
            SYSBERRIV = 0; //clear bus error flag
            USB_disable(); //Disable
    }
}
//Released_Version_5_20_06_02
```

init.c

```
/*
 * init.c
 *
 *  Created on: Nov 30, 2016
 *      Author: kdolgikh
 */

/** \file init.c
 *     \brief Contains initialization functions
 */

#include "driverlib.h"
#include "init.h"
#include "adc_driver.h"
#include "mux_adg706_driver.h"
#include "ucs_functions.h"
#include "hal_SPI.h"
#include "sdcard.h"
#include "Radio/CC110l.h"
#include "Radio/SPI_Library.h"
#include "sleep_timer.h"

void clocks_init (void)
{
       P1SEL |= BIT0;        // Set pins to output ACLK
       P1DIR |= BIT0;        // ACLK output at port 1.0
       P3SEL |= BIT4;        // Set pins to output SMCLK
       P3DIR |= BIT4;        // SMCLK output at port 3.4
       UCSCTL1 = DCORSEL_5;  // For MCLK of 20 MHz

       // Set FLLD divide by 2 and FLLN to 303 to provide 20 MHz MCLK
       UCSCTL2 = FLLD_1 + FLLN0 + FLLN1 + FLLN2 + FLLN3 + FLLN5 + FLLN8;

       // Set sources to DCOCLK; ACLK is sourced from XT1 by default
       UCSCTL4 = SELS__DCOCLK + SELM__DCOCLK;

       BAKCTL |= BAKDIS;     // Disable battery backup system
       enable_XT1_RTC();     // Enable XT1 crystal and RTC
}
```

```
void TA2_init (void)
{
        TA2CTL |= TASSEL_1;   // TA clock source is ACLK
        TA2CTL |= ID_3;       // Divide ACLK by 8 . Overflow is 16 seconds
        TA2CTL |= TACLR;      // Clear TA1R, see p 465 UG
        TA2CTL |= TAIE;       // Enable TAIFG interrupt
        TA2CTL &= ~TAIFG;     // Clear TAIFG
        TA2CCTL1 |= CCIE;     // Enable CCIFG interrupt
        TA2CCTL1 &= ~CCIFG;   // Clear CCIFG
}

// All unused ports output direction, low to reduce power consumption
void ports_init (void)
{
        P4OUT &= ~BIT6;       // Power MUX pin Low -> power from the battery
        P4DIR |= BIT6;        // Power MUX pin - output direction
        P4OUT &= ~BIT7;       // SD-card MUX ON pin Low -> power not applied to SD-card
        P4DIR |= BIT7;        // SD-card power MUX ON pin - output direction

        // Unused pins port 1 (and RF GDO0, GDO2)
        P1DIR |= BIT1|BIT2|BIT4|BIT6|BIT7;          // Output direction
        P1OUT &= ~(BIT1|BIT2|BIT4|BIT6|BIT7);       // Set low

        // Unused pins port 2 - all pins are unused
        P2DIR = 0xFF;  // Set all pins as output
        P2OUT = 0x00;  // Set all pins low

        // Unused pins port 3
        P3DIR |= BIT0|BIT1|BIT2|BIT3|BIT5|BIT6|BIT7;       // Output direction
        P3OUT &= ~(BIT0|BIT1|BIT2|BIT3|BIT5|BIT6|BIT7);    // Set low

        // Unused pins port 5
        P5DIR |= BIT0|BIT1|BIT4|BIT5|BIT6|BIT7;
        P5OUT &= ~(BIT0|BIT1|BIT4|BIT5|BIT6|BIT7);

        // Unused pins port 6
        P6DIR = 0xFF;  // Set all pins as output
        P6OUT = 0x00;  // Set all pins low

        // Unused pins port 7
        P7DIR |= BIT4|BIT5|BIT6|BIT7;
        P7OUT &= ~(BIT4|BIT5|BIT6|BIT7);

        // Unused pins port 9
        P9DIR |= BIT4|BIT5|BIT6|BIT7;
        P9OUT &= ~(BIT4|BIT5|BIT6|BIT7);
}

void user_button_init(void)
{
    P1DIR &= ~BIT3;   //input direction //should be 0 by default, page 426 UG
    P1OUT |= BIT3;    //pull-up is configured
    P1REN |= BIT3;    //pull-up is enabled
    P1IES |= BIT3;    //high-to-low transition
    //writing to P1OUT, P1DIR, P1REN (page 412 UG), P1IES (page 413 UG) can result in P1IFG
    //Hence, clear P1IFG before enabling port interrupt with P1IE
    P1IFG &= ~BIT3;   // set P1IFG to 0
    P1IE |= BIT3;     //enable interrupt
}
```

```
void Radio_Init(void)
{
       SPI_Strobe(SRES, Get_RX_FIFO);      // Reset radio
       Sleep_Timer_Fast(5);
       SPI_Send(IOCFG0,0x2E);              //Set GDO_RX to tri-state
       SPI_Send(IOCFG1,0x2E);
       SPI_Send(IOCFG2,0x2E);
       SPI_Strobe(SPWD, Get_RX_FIFO);      // Send radio to sleep
}

void disable_SVSM (void)
{
       PMMCTL0_H = PMMPW_H;          // Enter password to change PMM registers
       PMMRIE &= ~(SVSHPE | SVSLPE); // Disable SVS low and high side interrupts
       SVSMHCTL &= ~(SVMHE | SVSHE); // Disable high side SVS-SVM
       SVSMLCTL &= ~(SVMLE | SVSLE); // Disable low side SVS-SVM
       PMMCTL0_H = 0;                // Clear PMM password
}

void device_init(void)
{
       WDTCTL = WDTPW | WDTHOLD;           // Stop watchdog timer
       PMM_setVCore(PMM_CORE_LEVEL_3);     // Set VCore level to 3 to support 20 MHz MCLK
       clocks_init();                      // Set up clocks
       ports_init();                       // Initialize ports including unused ports
       TA2_init();                         // Set up TA2 for sleep_timer_fast
       adc_spi_init();                     // Init ADC pins and SPI
       SPI_Init();                         // Init radio SPI
       mux_adg706_ports_init();            // Init MUX ADG706 pins
       sdc_CS_Init();                      // Initialize uSD-card CS pin
       user_button_init();                 // P1.3 init
       disable_SVSM();                     // Disable SVS, SVM high and low sides
}
```

init.h

```
/*
 * init.h
 *
 *  Created on: Nov 30, 2016
 *      Author: kdolgikh
 */

/** \file init.h
 *     \brief Contains initialization function prototypes
 */

#ifndef INIT_H_
#define INIT_H_

/** \brief Perform clocks setup
 *
 *     Execution flow:
 *     1. Set ports to output ACLK and SMCLK.
 *     2. Select DCO frequency range to match specific MCLK frequency
 *     3. Set FLL divider and multiplier values to produce specific MCLK frequency
 *     4. Choose sources for ACLK, SMCLK and MCLK.
 *     5. Clear oscillator fault flags. See NOTE.
 *     6. Enable oscillator fault interrupts for the event log.
 *
```

```
 *      NOTE. According to MSP430F5659 Errata UCS12, changing the SELM/SELS/SELA bits
 *      in the UCSCTL4 register may set XT1/XT2 fault flags, which are needed to be
 *      cleared by user software.
 */
void clocks_init (void);

void TA2_init (void);

void ports_init (void);

/** \brief Performs device initialization
 *
 *  Execution flow:
 *  1. Disable watchdog timer.
 *  2. Enable XT1.
 *  3. Set up VCORE level to 3 to support 20 MHz MCLK.
 *
 */
void device_init (void);

void Radio_Init(void);

void disable_SVSM(void);

#endif /* INIT_H_ */
```

adc_driver.c

```
/*
 * adc_driver.c
 *
 *  Created on: Dec 2, 2016
 *      Author: kdolgikh
 */

/**
 *      \file adc_driver.c
 *      \brief TI ADS1247 driver source file
 */

#include "driverlib.h"
#include "adc_driver.h"
#include "states.h"
#include "stdint.h"
#include "crc8.h"
#include "USB_API/USB_Common/usb.h"
#include "send_over_usb.h"
#include "sleep_timer.h"
#include "data_prep.h"
#include "mux_adg706_driver.h"
#include "measure.h"

uint8_t adc_data_rate = ADC_DR_10SPS;
uint8_t adc_mux1_vrefcon = MUX1_VREFCON_VREF_ALT;

//ADC commands to read a single register
uint8_t adc_rreg_mux0[SINGLE_REG_RW_CMD_LGTH]  = {ADC_RREG_MUX0_BYTE1, ADC_BYTE2_SINGLE_REG,
ADC_NOP};
uint8_t adc_rreg_vbias[SINGLE_REG_RW_CMD_LGTH] = {ADC_RREG_VBIAS_BYTE1, ADC_BYTE2_SINGLE_REG,
ADC_NOP};
```

```
uint8_t adc_rreg_mux1[SINGLE_REG_RW_CMD_LGTH]  = {ADC_RREG_MUX1_BYTE1, ADC_BYTE2_SINGLE_REG,
ADC_NOP};
uint8_t adc_rreg_sys0[SINGLE_REG_RW_CMD_LGTH]  = {ADC_RREG_SYS0_BYTE1, ADC_BYTE2_SINGLE_REG,
ADC_NOP};
uint8_t adc_rreg_idac0[SINGLE_REG_RW_CMD_LGTH] = {ADC_RREG_IDAC0_BYTE1, ADC_BYTE2_SINGLE_REG,
ADC_NOP};
uint8_t adc_rreg_idac1[SINGLE_REG_RW_CMD_LGTH] = {ADC_RREG_IDAC1_BYTE1, ADC_BYTE2_SINGLE_REG,
ADC_NOP};
uint8_t adc_rreg_gpiocfg[SINGLE_REG_RW_CMD_LGTH] = {ADC_RREG_GPIOCFG_BYTE1,
ADC_BYTE2_SINGLE_REG, ADC_NOP};
uint8_t adc_rreg_gpiodir[SINGLE_REG_RW_CMD_LGTH]   = {ADC_RREG_GPIODIR_BYTE1,
ADC_BYTE2_SINGLE_REG, ADC_NOP};
uint8_t adc_rreg_gpiodat[SINGLE_REG_RW_CMD_LGTH]= {ADC_RREG_GPIODAT_BYTE1,
ADC_BYTE2_SINGLE_REG, ADC_NOP};

//ADC command to reed the first four registers
uint8_t adc_rreg_first4[FOUR_REG_RW_CMD_LGTH] = {ADC_RREG_MUX0_BYTE1, ADC_BYTE2_FOUR_REG,
ADC_NOP, ADC_NOP, ADC_NOP, ADC_NOP};

// ADC commands to read calibration registers
uint8_t adc_rreg_ofc[THREE_REG_RW_CMD_LGTH] = {ADC_RREG_OFC0_BYTE1, ADC_BYTE2_THREE_REG,
ADC_NOP, ADC_NOP, ADC_NOP};
uint8_t adc_rreg_fsc[THREE_REG_RW_CMD_LGTH] = {ADC_RREG_FSC0_BYTE1, ADC_BYTE2_THREE_REG,
ADC_NOP, ADC_NOP, ADC_NOP};

// ADC command to read all registers
const uint8_t adc_rreg_all[ALL_REG_R_CMD_LGTH]     =
{ADC_RREG_MUX0_BYTE1, ADC_BYTE2_ALL_REG,ADC_NOP, ADC_NOP, ADC_NOP, ADC_NOP, ADC_NOP,
ADC_NOP, ADC_NOP, ADC_NOP, ADC_NOP, ADC_NOP,ADC_NOP, ADC_NOP, ADC_NOP, ADC_NOP, ADC_NOP
};

const struct   // The struct with default values of ADC registers
{
        uint8_t adc_mux0;
        uint8_t adc_vbias;
        uint8_t adc_mux1;
        uint8_t adc_sys0;
        uint8_t adc_ofc0;
        uint8_t adc_ofc1;
        uint8_t adc_ofc2;
        uint8_t adc_fsc0;
        uint8_t adc_fsc1;
        uint8_t adc_fsc2;
        uint8_t adc_idac0;
        uint8_t adc_idac1;
        uint8_t adc_gpiocfg;
        uint8_t adc_gpiodir;
        uint8_t adc_gpiodat;
} adc_regs_default = {
        0x01,   // mux0
        0x00,   // vbias
        0x00,   // mux1
        0x00,   // sys0
        0x00,   // ofc0
        0x00,   // ofc1
        0x00,   // ofc2
        0x00,   // fsc0
        0x10,   // fsc1
        0x40,   // fsc2
        0x90,   // idac0
        0xFF,   // idac1
```

```
        0x00,   // gpiocfg
        0x00,   // gpiodir
        0x00    // gpiodat
};
uint8_t adc_calibration_register[CAL_REG_LGTH] = {0}; // Storage for a calibration register
uint8_t adc_regs_four[FOUR_REG_LGTH] = {0};           // Storage for the first four registers
mux0,vbias, mux1, sys0
uint8_t adc_regs_all[ADC_NUM_REGISTERS] = {0};        // Storage for all ADC registers
uint8_t sysocal_enable = FALSE;
uint8_t adc_wreg_single[SINGLE_REG_RW_CMD_LGTH] = {0};// Array for any WREG command

const uint8_t adc_wreg_sysgcal = ADC_SYSGCAL;         // Write system gain calibration command
const uint8_t adc_wreg_selfocal = ADC_SELFOCAL;       // Write self offset calibration command
const uint8_t adc_wreg_sysocal = ADC_SYSOCAL;         // Write system offset cal command

uint8_t adc_conv_result [CONV_LENGTH] = {0};          // ADC conversion result of 24 bits

adc_mux0_register *adc_mux0_ptr;                      // Ptr to struct of type adc_mux0_reg
adc_mux1_register *adc_mux1_ptr;                      // Ptr to struct of type adc_mux1_reg
adc_sys0_register *adc_sys0_ptr;                      // Ptr to struct of type adc_sys0_reg

uint8_t adcWorkInProgress = FALSE;                    // Flag that locks ADC operation

void adc_spi_pins_init (void)
{
        P5DIR |= ADC_CS_N + ADC_RST_N; // Set P5.2(ADC_CS*) and P5.3(ADC_RST*) output direction
        P5OUT |= ADC_CS_N + ADC_RST_N; // Set P5.2 (ADC_CS*) P5.3 (ADC_RST*) HIGH
        P9DIR |= ADC_START; // Set P9.0 (ADC_START) output direction
        P9OUT &= ~ADC_START;// Set P9.0 (ADC_START) LOW; PXOUT is not initialized upon reset
        P9SEL |= BIT1 + BIT2 + BIT3; // Select USCI function for ports P9.1, 9.2, 9.3
        P1OUT |= BIT5; // P1.5 (ADC_DRDY*) is initialized as input direction after reset. Set a
pull-up resistor for P1.5
        P1REN |= BIT5;  // Pull-up resistor is enabled
        P1IES |= BIT5;  // Select interrupt on a falling edge for P1.5 (ADC_DRDY*)
        P1IFG &= ~BIT5; // Clear P1.5 IFG
        P1IE |= BIT5;   // Enable interrupts for P1.5 (ADC_DRDY*)
}

void adc_spi_init (void)
{
        UCA2CTL1 |= UCSWRST; // Hold USCI in reset. Set by default
        //When set, the UCSWRST bit resets the UCRXIE, UCTXIE,
        //UCRXIFG, UCOE, and UCFE bits, and sets the UCTXIFG flag; page 972 UG

        UCA2CTL0 |= UCMSB;    // MSB first to interface with ADS1247
        UCA2CTL0 |= UCMST;    // Master mode
        UCA2CTL0 |= UCSYNC;   // Synchronous mode (SPI)
        //After reset, UCA2CTL0 default values are: 8 bit length, 3-pin SPI
        //ADS1247 uses Mode0 (polarity and phase are 0, as by default)

        UCA2CTL1 |= UCSSEL__SMCLK;   // SMCLK of 20 MHz is a source for SPI clock
        UCA2BR0 = 0x0A;              // Divide SMCLK by 10 to get 2 MHz SCLK. Max ADS1247 SCLK
is 2.05 MHz, page 10 ADS1247 UG
        UCA2BR1 = 0;                 // Set to 0 to use only UCA2BR0 value
        adc_spi_pins_init();         // Initialize ADC SPI pins
        UCA2CTL1 &= ~UCSWRST;        // Release USCI for operation
}
```

```
device_state adc_reset (void)
{
	P5OUT &= ~ADC_RST_N;
	__delay_cycles(40); // Duration of the RST pulse
	P5OUT |= ADC_RST_N;

	adcWorkInProgress = TRUE;
	Sleep_Timer_Fast(3); // Settling delay of 0.7 ms (should be greater than 0.6 ms)
	adcWorkInProgress = FALSE;

	if(!USBconnected)
		return adc_set;
	else
		return usb_connected;
}

uint8_t adc_spi_rreg_single (uint8_t *adc_rreg_cmd)
{
	uint8_t i;
	uint8_t adc_register = 0;

	P5OUT &= ~ADC_CS_N; // Set ADC_CS* LOW to enable SPI communication with ADC

	for (i = 0; i <= COMMAND_BYTE_3; i++)
	{
		while (!(UCA2IFG & UCTXIFG));		// Wait until TX is done
		UCA2TXBUF = *(adc_rreg_cmd + i);	// Send command bytes
		while (!(UCA2IFG & UCRXIFG));		// Wait until RX is done
		adc_register = UCA2RXBUF;		// Only the third received byte is required
	}
	__delay_cycles(30);				// Make sure Tsccs > 1.7 us

	P5OUT |= ADC_CS_N; // Set ADC_CS* HIGH to disable SPI communication with ADC

	return adc_register;
}

void adc_spi_rreg_first4 (uint8_t *adc_rreg_cmd, uint8_t *four_reg_stg)
{
	uint8_t i;
	P5OUT &= ~ADC_CS_N; // Set ADC_CS* LOW to enable SPI communication with ADC

	for (i = 0; i <= COMMAND_BYTE_6; i++)
	{
		while (!(UCA2IFG & UCTXIFG));		// Wait until TX is done
		UCA2TXBUF = *(adc_rreg_cmd + i);	// Send command bytes
		while (!(UCA2IFG & UCRXIFG));		// Wait until RX is done
		if (i < COMMAND_BYTE_3)
			*four_reg_stg = UCA2RXBUF;	// Discard the first two bytes
		else
		{
			*four_reg_stg = UCA2RXBUF;	// Store bytes 3-6 into the storage array
			four_reg_stg++;
		}
	}
	__delay_cycles(30); // Make sure Tsccs > 1.7 us

	P5OUT |= ADC_CS_N; // Set ADC_CS* HIGH to disable SPI communication with ADC
}
```

```
void adc_spi_rreg_cal_register (uint8_t *cal_reg_cmd, uint8_t *adc_cal_register)
{
	uint8_t i;

	P5OUT &= ~ADC_CS_N; // Set ADC_CS* LOW to enable SPI communication with ADC

	for (i = 0; i <= COMMAND_BYTE_5; i++)
	{
		while (!(UCA2IFG & UCTXIFG));		// Wait until TX is done
		UCA2TXBUF = *(cal_reg_cmd + i);		// Send command bytes
		while (!(UCA2IFG & UCRXIFG));		// Wait until RX is done
		if (i < COMMAND_BYTE_3)
			*adc_cal_register = UCA2RXBUF; // Discard the first two bytes
		else
		{
			*adc_cal_register = UCA2RXBUF; // Store bytes 3-5
			adc_cal_register++;
		}
	}
	__delay_cycles(30); // Make sure Tsccs > 1.7 us

	P5OUT |= ADC_CS_N;
}

void adc_spi_rreg_all (uint8_t *adc_regs_all)
{
	uint8_t i;

	P5OUT &= ~ADC_CS_N; // Set ADC_CS* LOW to enable SPI communication with ADC

	for (i = 0; i <= COMMAND_BYTE_17; i++)
	{
		while (!(UCA2IFG & UCTXIFG));		// Wait until TX is done
		UCA2TXBUF = *(adc_rreg_all + i);	// Send command bytes
		while (!(UCA2IFG & UCRXIFG));		// wait until RX is done
		if (i < COMMAND_BYTE_3)
			*adc_regs_all = UCA2RXBUF;	// Discard the first two bytes
		else
		{
			*adc_regs_all = UCA2RXBUF;	// Store bytes 3-17
			adc_regs_all++;
		}
	}
	__delay_cycles(30); // Make sure Tsccs > 1.7 us

	P5OUT |= ADC_CS_N; // Set ADC_CS* HIGH to disable SPI communication with ADC
}

void adc_spi_define_wreg_mux0 (adc_mux0_register *adc_mux0_param)
{
	uint8_t mux0_register;
	mux0_register =		// Define the contents of the register
		adc_mux0_param->adc_bcs +
		adc_mux0_param->adc_mux_pos_input +
		adc_mux0_param->adc_mux_neg_input;
	*adc_wreg_single = ADC_WREG_MUX0_BYTE1; // Write starting from MUX0 register
	*(adc_wreg_single + COMMAND_BYTE_2) = ADC_BYTE2_SINGLE_REG; // Write only one register
at a time
	*(adc_wreg_single + COMMAND_BYTE_3) = mux0_register; // Data to be written to the
register
}
```

```
void adc_spi_define_wreg_mux1 (adc_mux1_register *adc_mux1_param)
{
	uint8_t mux1_register;
	mux1_register =	// Define the contents of the register
		adc_mux1_param->adc_clkstat +
		adc_mux1_param->adc_vrefcon +
		adc_mux1_param->adc_refselt +
		adc_mux1_param->adc_muxcal;

	*adc_wreg_single = ADC_WREG_MUX1_BYTE1; // Write starting from MUX1 register
	*(adc_wreg_single + COMMAND_BYTE_2) = ADC_BYTE2_SINGLE_REG; // Write only one register
at a time
	*(adc_wreg_single + COMMAND_BYTE_3) = mux1_register; // Data to be written to the
register
}

void adc_spi_define_wreg_sys0 (adc_sys0_register *adc_sys0_param)
{
	uint8_t sys0_register;
	sys0_register =	// Define the contents of the register
		adc_sys0_param->adc_pga_gain +
		adc_sys0_param->adc_data_rate;

	*adc_wreg_single = ADC_WREG_SYS0_BYTE1; // Write starting from SYS0 register
	*(adc_wreg_single + COMMAND_BYTE_2) = ADC_BYTE2_SINGLE_REG; // Write only one register
at a time
	*(adc_wreg_single + COMMAND_BYTE_3) = sys0_register; // Data to be written to the
register
}

// Need to make this function generic, applicable to any register and not just three
uint8_t adc_spi_wreg_single (void)
{
	uint8_t i;
	uint8_t adc_reg = 0;

	P5OUT &= ~ADC_CS_N; // Set ADC_CS* LOW to enable SPI communication with ADC

	for (i = 0; i <= COMMAND_BYTE_3; i++)
	{
		while (!(UCA2IFG & UCTXIFG));	// Wait until TX is done
		UCA2TXBUF = *(adc_wreg_single + i); // Send command bytes
		while (!(UCA2IFG & UCRXIFG));	// Wait until RX is done
		adc_reg = UCA2RXBUF; // Read from RX buffer to clear the RXIFG
	}
	__delay_cycles(30); // Make sure Tsccs > 1.7 us

	if (*adc_wreg_single == ADC_WREG_MUX0_BYTE1) // Does the 1st element equal to MUX0 cmd?
		adc_reg = adc_spi_rreg_single (adc_rreg_mux0); // If yes, read MUX0 register
	else if (*adc_wreg_single == ADC_WREG_MUX1_BYTE1) // Does the 1st element = MUX1 cmd?
		adc_reg = adc_spi_rreg_single (adc_rreg_mux1); // If yes, read MUX1 register
	else if (*adc_wreg_single == ADC_WREG_SYS0_BYTE1) // Does 1st element = SYS0 cmd?
		adc_reg = adc_spi_rreg_single (adc_rreg_sys0); // If yes, read SYS0 register

	if (adc_reg == *(adc_wreg_single + COMMAND_BYTE_3)) // Check if read value = written
value
		return WRITE_SUCCESS;
	else
		return WRITE_FAIL;
}
```

```
uint8_t adc_spi_wreg_mux0 (void)
{
	adc_mux0_register adc_mux_0 =
	{
		adc_mux_0.adc_bcs = MUX0_BCS_OFF,
		adc_mux_0.adc_mux_pos_input = SP_AIN2, // invert the inputs from the default
		adc_mux_0.adc_mux_neg_input = SN_AIN3
	};
	adc_mux0_ptr = &adc_mux_0;
	adc_spi_define_wreg_mux0(adc_mux0_ptr);
	return adc_spi_wreg_single();
}

uint8_t adc_spi_wreg_mux1_sys0 (adc_mux1_register *adc_mux1_param,
                                adc_sys0_register *adc_sys0_param)
{
	uint8_t adc_w_cmd[TWO_REG_RW_CMD_LGTH] = {0}; // write command
	uint8_t mux1_register;
	uint8_t sys0_register;
	uint8_t i;
	uint8_t adc_reg = 0;
	mux1_register =       // Define the contents of the register
			adc_mux1_param->adc_clkstat +
			adc_mux1_param->adc_vrefcon +
			adc_mux1_param->adc_refselt +
			adc_mux1_param->adc_muxcal;

	sys0_register =       // Define the contents of the register
			adc_sys0_param->adc_pga_gain +
			adc_sys0_param->adc_data_rate;

	*adc_w_cmd = ADC_WREG_MUX1_BYTE1; // Write starting from MUX1 register
	*(adc_w_cmd + COMMAND_BYTE_2) = ADC_BYTE2_TWO_REG; // Write two registers at once
	*(adc_w_cmd + COMMAND_BYTE_3) = mux1_register; // Data to be written to the register
	*(adc_w_cmd + COMMAND_BYTE_4) = sys0_register; // Data to be written to the register

	P5OUT &= ~ADC_CS_N; // Set ADC_CS* LOW to enable SPI communication with ADC
	for (i = 0; i <= COMMAND_BYTE_4; i++)
	{
		while (!(UCA2IFG & UCTXIFG));        // Wait until TX is done
		UCA2TXBUF = *(adc_w_cmd + i);        // Send command bytes
		while (!(UCA2IFG & UCRXIFG));        // Wait until RX is done
		adc_reg = UCA2RXBUF;                 // Read from RX buffer to clear the RXIFG
	}
	__delay_cycles(30); // Make sure Tsccs > 1.7 us
	P5OUT |= ADC_CS_N;
	__delay_cycles(20); // Make sure Tcspw > 1.22 us

	adc_reg = adc_spi_rreg_single (adc_rreg_mux1);
	if (adc_reg == mux1_register)
	{
		__delay_cycles(20); // Make sure Tcspw > 1.22 us
		adc_reg = adc_spi_rreg_single (adc_rreg_sys0);
		if (adc_reg == sys0_register)
			return WRITE_SUCCESS;
		else
			return WRITE_FAIL;
	}
	else return WRITE_FAIL;
}
```

```
uint8_t adc_spi_wreg_first4 (adc_mux0_register *adc_mux0_param,
                             adc_mux1_register *adc_mux1_param,
                             adc_sys0_register *adc_sys0_param)
{
        uint8_t adc_w_cmd[FOUR_REG_RW_CMD_LGTH] = {0};     // Write command
        uint8_t mux0_register;
        uint8_t vbias_register = 0;                        // vbias register is 0
        uint8_t mux1_register;
        uint8_t sys0_register;
        uint8_t i;
        uint8_t adc_reg = 0;

        mux0_register =       // Define the contents of the register
                        adc_mux0_param->adc_bcs +
                        adc_mux0_param->adc_mux_pos_input +
                        adc_mux0_param->adc_mux_neg_input;

        mux1_register =       // Define the contents of the register
                        adc_mux1_param->adc_clkstat +
                        adc_mux1_param->adc_vrefcon +
                        adc_mux1_param->adc_refselt +
                        adc_mux1_param->adc_muxcal;

        sys0_register =       // Define the contents of the register
                              adc_sys0_param->adc_pga_gain +
                              adc_sys0_param->adc_data_rate;

        *adc_w_cmd = ADC_WREG_MUX0_BYTE1; // Write starting from MUX0 register
        *(adc_w_cmd + COMMAND_BYTE_2) = ADC_BYTE2_FOUR_REG; // Write four registers at once
        *(adc_w_cmd + COMMAND_BYTE_3) = mux0_register; // Data to be written to the register
        *(adc_w_cmd + COMMAND_BYTE_4) = vbias_register;// Data to be written to the register
        *(adc_w_cmd + COMMAND_BYTE_5) = mux1_register; // Data to be written to the register
        *(adc_w_cmd + COMMAND_BYTE_6) = sys0_register; // Data to be written to the register

        P5OUT &= ~ADC_CS_N; // Set ADC_CS* LOW to enable SPI communication with ADC

        for (i = 0; i <= COMMAND_BYTE_6; i++)
        {
                while (!(UCA2IFG & UCTXIFG)); // Wait until TX is done
                UCA2TXBUF = *(adc_w_cmd + i); // Send command bytes
                while (!(UCA2IFG & UCRXIFG)); // Wait until RX is done
                adc_reg = UCA2RXBUF;          // Read from RX buffer to clear the RXIFG
        }
        __delay_cycles(30);                   // Make sure Tsccs > 1.7 us
        P5OUT |= ADC_CS_N;
        __delay_cycles(20);                   // Make sure Tcspw > 1.22 us

        adc_spi_rreg_first4(adc_rreg_first4, adc_regs_four); // Read the first four registers

        if ((*adc_regs_four == mux0_register) &&
                        (*(adc_regs_four + 1) == vbias_register) &&
                        (*(adc_regs_four + 2) == mux1_register) &&
                        (*(adc_regs_four + 3) == sys0_register))
                return WRITE_SUCCESS;
        else
                return WRITE_FAIL;
}
```

```
device_state adc_setup (uint8_t data_rate, uint8_t mux1_vrefcon)
{
	uint8_t status = 0;
	uint8_t num_write = 0;
	adc_mux0_register adc_mux_0 =
	{
		adc_mux_0.adc_bcs = MUX0_BCS_OFF,
		adc_mux_0.adc_mux_pos_input = SP_AIN2, //SP_AIN2
		adc_mux_0.adc_mux_neg_input = SN_AIN3  //SN_AIN3
	};

	adc_mux1_register adc_mux_1 =
	{
		adc_mux_1.adc_clkstat = MUX1_CLKSTAT_INT_OSC, // Internal oscillator
		adc_mux_1.adc_vrefcon = mux1_vrefcon,     // Control the VREF module
		adc_mux_1.adc_refselt = MUX1_REFSELT_INT, // VREF is the source for ref voltage
		adc_mux_1.adc_muxcal = MUX1_MUXCAL_NORM   // Normal operation
	};
	adc_sys0_register adc_sys_0 =
	{
		adc_sys_0.adc_pga_gain = ADC_GAIN_1,
		adc_sys_0.adc_data_rate = data_rate
	};

	adc_mux0_ptr = &adc_mux_0;
	adc_mux1_ptr = &adc_mux_1;
	adc_sys0_ptr = &adc_sys_0;

	adcWorkInProgress = TRUE; // Lock device for ADC operation

	P9OUT |= ADC_START; // START pin must be taken high before communicating with registers

	do
	{
		status = adc_spi_wreg_first4(adc_mux0_ptr, adc_mux1_ptr, adc_sys0_ptr); // Write
the first 4 registers simultaneously to avoid 64 clocks requirement between individual
register writes
		num_write++;
	}
	while ((status == WRITE_FAIL) && (num_write <= NUM_WRITE)); // If retrying, Tcspw is ok

	P9OUT &= ~ADC_START; // START pin low to put ADC into sleep mode after the conversion
is finished

	LPM3; // Wake upon DRDY* interrupt, the conversion result is discarded

	adcWorkInProgress = FALSE; // Release device for other operations

	if(!USBconnected)
	{
		if (status == WRITE_SUCCESS)
			return adc_measuring;
		else
			return dev_malfunction;
	}
	else
	{
		usb_exit=exit3;
		return usb_connected;
	}
}
```

```
void adc_spi_rdata_once (void)
{
	uint8_t i;

	P5OUT &= ~ADC_CS_N; // Set ADC_CS* LOW to enable SPI communication with ADC

	for (i=0; i < CONV_LENGTH; i++)
	{
		while (!(UCA2IFG & UCTXIFG));		// Wait until TX is done
		UCA2TXBUF = ADC_NOP;			// Send NOP
		while (!(UCA2IFG & UCRXIFG));		// Wait until RX is done
		*(adc_conv_result + i) = UCA2RXBUF; // Read from RX buffer to clear the RXIFG
	}
	__delay_cycles(30); // Make sure Tsccs > 1.7 us

	P5OUT |= ADC_CS_N; // Set ADC_CS* HIGH to disable SPI communication with ADC
}

uint16_t adc_measure (uint16_t num_conv, // num_conv depends on RAM storage size.
		      uint8_t *storage,  // RAM storage for conversion results in bytes.
		      uint16_t index)    // for the very first conversion index should be 0
{
	uint8_t k;
	uint8_t i;

// The conversion results are read in continuous mode (START is high for the duration of
measurement)
	P9OUT |= ADC_START; // START conversion

	for (i=0; i < num_conv; i++)
	{
		if (i == (num_conv - 1))	// Check if last conversion
		{
			__delay_cycles(30);	// Make sure min Tstart > 3tosc or 0.73 us
			P9OUT &= ~ADC_START;	// START pin should be set low before the last
conversion's DRDY* interrupt
		}
		LPM3; // Wait until current conversion is over, will return from DRDY* ISR

		adc_spi_rdata_once();		// Retrieve the conversion result

		for (k=0; k < CONV_LENGTH; k++) // Store result in RAM storage
		{
			*(storage + index) = *(adc_conv_result + k);
			index++; // Next time the first conv result byte will be stored in the
third element of RAM storage
		}
	}
	return index; // If this function is used for one conversion at a time,
		      // then it will communicate info about the index in ram_stg
		      // where to store next conversion in the next function call.
}

uint16_t adc_measure_all_channels(uint8_t *storage, uint16_t index, uint16_t *crc)
{
	uint8_t k;

	adcWorkInProgress = TRUE;

	for (k = 0; k < NUM_CHANNELS; k++) // Repeat measurement for 16 channels
    {
```

```
        switch_mux_ch(k+1);

        index = adc_measure(SINGLE_CONV,
                            storage,
                            index);
    }

       adcWorkInProgress = FALSE;

       disable_mux_adg706(); // Disable mux to conserve power

       *crc = crc8_set((storage+index-CONV_LENGTH_16CH),
                       CONV_LENGTH_16CH); // Generate CRC of 16 measurements

    int_divide(*crc,(storage+index)); // Divide crc_value into two bytes, store in RAM storage
after 16 measurements

    index = index + CRC_NUM_BYTES; // Index points to the element after CRC

    return index;
}

uint8_t adc_sysgcal (uint8_t data_rate)
{
       uint8_t status = 0;
       uint8_t num_write = 0;
       uint8_t adc_reg = 0;
       adc_mux1_register adc_mux_1 =
       {
              adc_mux_1.adc_clkstat = MUX1_CLKSTAT_INT_OSC,  // Internal oscillator
              adc_mux_1.adc_vrefcon = MUX1_VREFCON_VREF_ALT, // VREF is on/off when necessary
              adc_mux_1.adc_refselt = MUX1_REFSELT_INT_REF0, // VREF is connected to REF0
              adc_mux_1.adc_muxcal = MUX1_MUXCAL_GAIN_CAL    // Gain calibration
       };
       adc_sys0_register adc_sys_0 =
       {
              adc_sys_0.adc_pga_gain = ADC_GAIN_1,
              adc_sys_0.adc_data_rate = data_rate
       };
       adc_mux1_ptr = &adc_mux_1;
       adc_sys0_ptr = &adc_sys_0;

       adcWorkInProgress = TRUE;

       P9OUT |= ADC_START; // START pin must be taken high before communicating with registers

       do
       {
              status = adc_spi_wreg_mux1_sys0(adc_mux1_ptr, adc_sys0_ptr); // Write both mux1
and sys0 simultaneously
              num_write++;
       }
       while ((status == WRITE_FAIL) && (num_write <= NUM_WRITE)); // With retry Tcspw is
still ok

       if (status == WRITE_SUCCESS)
       {
              __delay_cycles(20); // Make sure Tcspw > 1.22 us
              P5OUT &= ~ADC_CS_N; // Set ADC_CS* LOW to enable SPI communication with ADC

              while (!(UCA2IFG & UCTXIFG)); // Wait until TX is done
```

```
                UCA2TXBUF = adc_wreg_sysgcal; // Send sysgcal command
            while (!(UCA2IFG & UCRXIFG)); // Wait until RX is done
                adc_reg = UCA2RXBUF;   // Read from RX buffer to clear the RXIFG
            __delay_cycles(30);           // Make sure that Tsccs > 1.7 us
            P5OUT |= ADC_CS_N;   // Set ADC_CS* HIGH to disable SPI communication with ADC
            P9OUT &= ~ADC_START; // Set START LOW places ADS into sleep after calibration

            LPM3; // Wait until calibration is done in LPM3, wake upon DRDY* interrupt

            adcWorkInProgress = FALSE;

            return STATUS_SUCCESS; // sysgcal is successfully accomplished
        }
        else
        {
            P9OUT &= ~ADC_START; // Set START LOW places ADS into sleep after calibration
            LPM3;                // Wake upon DRDY* interrupt

            adcWorkInProgress = FALSE;

            return STATUS_FAIL;
        }
}

void adc_selfocal (void)
{
        uint8_t adc_reg = 0;

        adcWorkInProgress = TRUE;

        P9OUT |= ADC_START; // START pin must be taken high before communicating with
registers, page 33 ADS1247 UGS
        P5OUT &= ~ADC_CS_N; // Set ADC_CS* LOW to enable SPI communication with ADC

        // send selfocal
        while (!(UCA2IFG & UCTXIFG)); // Wait until TX is done
            UCA2TXBUF = adc_wreg_selfocal; // Send command bytes
        while (!(UCA2IFG & UCRXIFG)); // Wait until RX is done
            adc_reg = UCA2RXBUF;   // Read from RX buffer to clear the RXIFG

        __delay_cycles(20);   // Make sure that Tsccs > 1.7 us

        P5OUT |= ADC_CS_N;    // Set ADC_CS* HIGH to disable SPI communication with ADC
        P9OUT &= ~ADC_START;  // Set START LOW to place ADC into sleep mode

        LPM3; // Wait until calibration is done in LPM3

        adcWorkInProgress = FALSE;
}

void adc_sysocal (void)
{
        uint8_t adc_reg = 0;

        adcWorkInProgress = TRUE;

        P9OUT |= ADC_START;   // START pin must be taken high before communicating with
registers, page 33 ADS1247 UGS
        P5OUT &= ~ADC_CS_N;   // Set ADC_CS* LOW to enable SPI communication with ADC

        // send sysocal
```

```
        while (!(UCA2IFG & UCTXIFG));         // Wait until TX is done
               UCA2TXBUF = adc_wreg_sysocal; // Send command bytes
        while (!(UCA2IFG & UCRXIFG));         // Wait until RX is done
               adc_reg = UCA2RXBUF;           // Read from RX buffer to clear the RXIFG
        __delay_cycles(20);                   // Make sure that Tsccs > 1.7 us

        P5OUT |= ADC_CS_N; // Set ADC_CS* HIGH to disable SPI communication with ADC
        P9OUT &= ~ADC_START; // Set START low places ADS to sleep mode when finish calibration

        LPM3; // Wait until calibration is done in LPM3

        adcWorkInProgress = FALSE;
}

device_state adc_calibration (uint8_t data_rate)
{
        uint8_t status = 0;

        status = adc_sysgcal(data_rate);  // System gain calibration

        if (!USBconnected)
        {
                if (status == STATUS_SUCCESS) // If successful, proceed to self offset
calibration
                {
                        if (sysocal_enable == TRUE)
                                adc_sysocal();  // System offset calibration
                        else
                                adc_selfocal(); // Self offset calibration

                        if (!USBconnected)
                                return adc_set;
                        else
                        {
                                usb_exit=exit2;
                                return usb_connected;
                        }
                }
                else
                        return dev_malfunction;
        }
        else
        {
                usb_exit=exit1;
                return usb_connected;
        }
}
```

adc_driver.h

```
/*
 * adc_driver.h
 *
 *  Created on: Dec 2, 2016
 *      Author: kdolgikh
 */

/**
 *      \file adc_driver.h
 *      \brief TI ADS1247 driver
```

```
 */

#ifndef ADC_DRIVER_H_
#define ADC_DRIVER_H_

#include "msp430.h"
#include "stdint.h"
#include "states.h"

/// \name Number of ADC conversions
//@{
#define SETTL_TIME                  4  // Num of Sleep_Timer_Fast cycles to make 1 ms of
settling time
#define ADC_NUM_REGISTERS           15 // Number of ADC registers
#define NUM_CHANNELS                16 // Number of ADC channels
#define NUM_WRITE                   2  // Number of ADC write register retries
//@}

/// \name Pins for SPI communication with ADC
//@{
#define ADC_CS_N  BIT2
#define ADC_RST_N BIT3
#define ADC_START BIT0
//@}

/// \name ADS1247 single byte commands
//@{
#define ADC_WAKEUP                  0x01
#define ADC_SLEEP                   0x02
#define ADC_RESET                   0x06
#define ADC_NOP                     0xFF
#define ADC_RDATA                   0x12
#define ADC_RDATAC                  0x14
#define ADC_SDATAC                  0x16
#define ADC_SYSOCAL                 0x60
#define ADC_SYSGCAL                 0x61
#define ADC_SELFOCAL                0x62
//@}

/// \name The first byte of the RREG commands:
//@{
#define ADC_RREG_MUX0_BYTE1         0x20 // start reading from the first register
#define ADC_RREG_VBIAS_BYTE1        0x21
#define ADC_RREG_MUX1_BYTE1         0x22
#define ADC_RREG_SYS0_BYTE1         0x23
#define ADC_RREG_OFC0_BYTE1         0x24
#define ADC_RREG_OFC1_BYTE1         0x25
#define ADC_RREG_OFC2_BYTE1         0x26
#define ADC_RREG_FSC0_BYTE1         0x27
#define ADC_RREG_FSC1_BYTE1         0x28
#define ADC_RREG_FSC2_BYTE1         0x29
#define ADC_RREG_IDAC0_BYTE1        0x2A
#define ADC_RREG_IDAC1_BYTE1        0x2B
#define ADC_RREG_GPIOCFG_BYTE1      0x2C
#define ADC_RREG_GPIODIR_BYTE1      0x2D
#define ADC_RREG_GPIODAT_BYTE1      0x2E // start reading from the last register
//@}

/**
 *      Note. The following registers don't require writing and stay unmodified:
 *      VBIAS, OFC0-2, FSC0-2, IDAC0-1, all GPIO registers.
```

```
 */
/// \name The first byte of the WREG commands
//@{
#define ADC_WREG_MUX0_BYTE1         0x40
#define ADC_WREG_MUX1_BYTE1         0x42
#define ADC_WREG_SYS0_BYTE1         0x43
//@}


/// \name The second byte of the RREG/WREG commands
//@{
#define ADC_BYTE2_SINGLE_REG        0x00 // read/write single register
#define ADC_BYTE2_TWO_REG           0x01 // read/write two registers
#define ADC_BYTE2_THREE_REG         0x02 // read/write three registers
#define ADC_BYTE2_FOUR_REG          0x03 // read/write four registers
#define ADC_BYTE2_ALL_REG           0x0D // read/write all registers
//@}

/// \name Lengths of a conversion result, commands and calibration registers
//@{
#define CONV_LENGTH                 3 // Length of a single conversion result, byte
#define CAL_REG_LGTH                3 // Length of a calibration register, byte
#define FOUR_REG_LGTH               4 // Length of three registers mux0, vbias, mux1, sys0
#define SINGLE_REG_RW_CMD_LGTH      3 // Length of the command to read/write a single
register, byte
#define TWO_REG_RW_CMD_LGTH         4 // Length of the command to read/write two registers,
byte
#define THREE_REG_RW_CMD_LGTH       5 // Length of the command to read/write three registers,
byte
#define FOUR_REG_RW_CMD_LGTH        6 // Length of the command to read/write four registers,
byte
#define ALL_REG_R_CMD_LGTH          17 // Length of the command to read all registers, byte
//@}

extern uint8_t adcWorkInProgress;

/**
 * ADC data rate
 *
 * Not real value but the value of the sys0 register that defines the data rate
 */
extern uint8_t adc_data_rate;

/**
 * Value of the ADC vrefcon field in mux1 register
 */
extern uint8_t adc_mux1_vrefcon;

/**
 * ADC commands to read a single register
 */
extern uint8_t adc_rreg_mux0[SINGLE_REG_RW_CMD_LGTH], adc_rreg_vbias[SINGLE_REG_RW_CMD_LGTH],
                            adc_rreg_mux1[SINGLE_REG_RW_CMD_LGTH],
                            adc_rreg_sys0[SINGLE_REG_RW_CMD_LGTH],
                            adc_rreg_idac0[SINGLE_REG_RW_CMD_LGTH],
                            adc_rreg_idac1[SINGLE_REG_RW_CMD_LGTH],
                            adc_rreg_gpiocfg[SINGLE_REG_RW_CMD_LGTH],
                            adc_rreg_gpiodir[SINGLE_REG_RW_CMD_LGTH],
                            adc_rreg_gpiodat[SINGLE_REG_RW_CMD_LGTH];

/**
```

```
 * ADC commands to read calibration registers
 */
extern uint8_t adc_rreg_ofc[THREE_REG_RW_CMD_LGTH],adc_rreg_fsc[THREE_REG_RW_CMD_LGTH];

/**
 * Storage for a calibration register
 */
extern uint8_t adc_calibration_register[CAL_REG_LGTH];

/**
 * If set, sysocal calibration is enabled in the calibration function
 */
extern uint8_t sysocal_enable;

/**
 * ADC command to read all registers
 */
extern const uint8_t adc_rreg_all[ALL_REG_R_CMD_LGTH];

/**
 * Storage for all ADC registers
 */
extern uint8_t adc_regs_all[ADC_NUM_REGISTERS];

/**
 * Array for any single write WREG command
 */
extern uint8_t adc_wreg_single[SINGLE_REG_RW_CMD_LGTH];

/**
 * Array for a single conversion
 */
extern uint8_t adc_conv_result [CONV_LENGTH];

/**
 * ADC calibration commands
 */
extern const uint8_t adc_wreg_sysgcal, adc_wreg_selfocal, adc_wreg_sysocal;

/// \name Positions of bytes for a single read/write and multiple read commands
//@{
#define COMMAND_BYTE_2              1
#define COMMAND_BYTE_3              2
#define COMMAND_BYTE_4              3
#define COMMAND_BYTE_5              4
#define COMMAND_BYTE_6              5
#define COMMAND_BYTE_17             16
//@}
/**
 *      A struct for ADS1247 MUX0 register
 *
 *      Note. For VE and CE schematics, only adc_bcs field might be used for a detection
 *      of a failed sensor. Analog inputs do not change.
 */
typedef struct adc_mux0_reg
{
        //!    The magnitude of the sensor detect current source
        //!    \n Valid values:
        //!    - \b MUX0_BCS_OFF             - burnout current source is off
        //!    - \b MUX0_BCS_05uA            - current source is on, 0.5 uA
        //!    - \b MUX0_BCS_2uA             - current source is on, 2 uA
```

```
        //!     - \b MUX0_BCS_10uA            - current source is on, 10 uA
        uint8_t adc_bcs;

        //!     Positive input channel
        //!     \n Valid values:
        //!     - \b AIN0                     - default
        //!     - \b AIN1
        //!     - \b AIN2
        //!     - \b AIN3
        uint8_t adc_mux_pos_input;

        //!     Negative input channel
        //!     \n Valid values:
        //!     - \b AIN0
        //!     - \b AIN1                     - default
        //!     - \b AIN2
        //!     - \b AIN3
        uint8_t adc_mux_neg_input;
} adc_mux0_register;

/// \name Valid values for the ADS1247 MUX0 register
//@{
#define MUX0_BCS_OFF                0x00
#define MUX0_BCS_05uA               0x40
#define MUX0_BCS_2uA                0x80
#define MUX0_BCS_10uA               0xC0
#define SP_AIN0                     0x00
#define SP_AIN1                     0x08
#define SP_AIN2                     0x10
#define SP_AIN3                     0x18
#define SN_AIN0                     0x00
#define SN_AIN1                     0x01
#define SN_AIN2                     0x02
#define SN_AIN3                     0x03
//@}

/**
 *      A struct for ADS1247 MUX1 register
 */
typedef struct adc_mux1_reg
{
        //!     Indicates the type of oscillator being used
        //!     \n Valid values:
        //!     - \b MUX1_CLKSTAT_INT_OSC     - Internal oscillator
        //!     - \b MUX1_CLKSTAT_EXT_OSC     - External oscillator
        uint8_t adc_clkstat;


        //!     Internal voltage reference control
        //!     \n Valid values:
        //!     - \b MUX1_VREFCON_VREF_OFF    - Internal reference is always off (default)
        //!     - \b MUX1_VREFCON_VREF_ON     - Internal reference is always on
        //!     - \b MUX1_VREFCON_VREF_ALT    - Int ref is on during conversion, is off when the
        //!     device receives a shutdown opcode or the START pin is taken low
        uint8_t adc_vrefcon;

        //!     Select the reference input for ADC
        //!     \n Valid values:
        //!     - \b MUX1_REFSELT_REF0        - REF0 input pair selected (default)
        //!     - \b MUX1_REFSELT_INT         - Internal reference selected
        //!     - \b MUX1_REFSELT_INT_REF0    - Int ref selected and connected to REF0 input
```

```
        uint8_t adc_refselt;

        //!     ADC calibration control
        //!     \n Valid values:
        //!     - \b MUX1_MUXCAL_NORM               - Normal operation (default)
        //!     - \b MUX1_MUXCAL_OFFSET_CAL         - Offset measurement
        //!     - \b MUXCAL_GAIN_CAL                - Gain measurement
        //!     - \b MUX1_MUXCAL_TEMP_MEAS          - Temperature diode
        //!     - \b MUX1_MUXCAL_EXT_REF0_MEAS      - External REF0 measurement
        //!     - \b MUX1_MUXCAL_AVDD_MEAS          - AVDD measurement
        //!     - \b MUX1_MUXCAL_DVDD_MEAS          - DVDD measurement
        uint8_t adc_muxcal;
} adc_mux1_register;

/// \name Valid values for the ADS1247 MUX1 register, see page 43 datasheet
//@{
#define MUX1_CLKSTAT_INT_OSC        0x00
#define MUX1_CLKSTAT_EXT_OSC        0x80
#define MUX1_VREFCON_VREF_OFF       0x00
#define MUX1_VREFCON_VREF_ON        0x20
#define MUX1_VREFCON_VREF_ALT       0x40
#define MUX1_REFSELT_REF0           0x00
#define MUX1_REFSELT_INT            0x10
#define MUX1_REFSELT_INT_REF0       0x18
#define MUX1_MUXCAL_NORM            0x00
#define MUX1_MUXCAL_OFFSET_CAL      0x01
#define MUX1_MUXCAL_GAIN_CAL        0x02
#define MUX1_MUXCAL_TEMP_MEAS       0x03
#define MUX1_MUXCAL_EXT_REF0_MEAS   0x05
#define MUX1_MUXCAL_AVDD_MEAS       0x06
#define MUX1_MUXCAL_DVDD_MEAS       0x07
//@}

/**
 *      A struct for ADS1247 SYS0 register
 */
typedef struct adc_sys0_reg
{
        //!     ADC PGA gain setting
        //!     \n Valid values:
        //!     - \b ADC_GAIN_1             - Gain equals 1
        //!     - \b ADC_GAIN_2             - Gain equals 2
        //!     - \b ADC_GAIN_4             - Gain equals 4
        //!     - \b ADC_GAIN_8             - Gain equals 8
        //!     - \b ADC_GAIN_16            - Gain equals 16
        //!     - \b ADC_GAIN_32            - Gain equals 32
        //!     - \b ADC_GAIN_64            - Gain equals 64
        uint8_t adc_pga_gain;

        //!     ADC output data rate setting
        //!     \n Valid values:
        //!     - \b ADC_DR_5SPS            - 5 SPS
        //!     - \b ADC_DR_10SPS           - 10 SPS
        //!     - \b ADC_DR_20SPS           - 20 SPS
        //!     - \b ADC_DR_40SPS           - 40 SPS
        //!     - \b ADC_DR_80SPS           - 80 SPS
        //!     - \b ADC_DR_160SPS          - 160 SPS
        //!     - \b ADC_DR_320SPS          - 320 SPS
        //!     - \b ADC_DR_640SPS          - 640 SPS
        //!     - \b ADC_DR_1000SPS         - 1000 SPS
        //!     - \b ADC_DR_2000SPS         - 2000 SPS
```

```
        uint8_t adc_data_rate;
} adc_sys0_register;

/// \name Valid values for the ADS1247 SYS0 register, gain setting
//@{
#define ADC_GAIN_1           0x00
#define ADC_GAIN_2           0x10
#define ADC_GAIN_4           0x20
#define ADC_GAIN_8           0x30
#define ADC_GAIN_16          0x40
#define ADC_GAIN_32          0x50
#define ADC_GAIN_64          0x60
//@}

/// \name Valid values for the ADS1247 SYS0 register, output data rate setting
//@{
#define ADC_DR_5SPS          0x00
#define ADC_DR_10SPS         0x01
#define ADC_DR_20SPS         0x02
#define ADC_DR_40SPS         0x03
#define ADC_DR_80SPS         0x04
#define ADC_DR_160SPS        0x05
#define ADC_DR_320SPS        0x06
#define ADC_DR_640SPS        0x07
#define ADC_DR_1000SPS       0x08
#define ADC_DR_2000SPS       0x09
//@}

/**
* Pointers to structures of type adc_mux0_reg, adc_mux1_reg and adc_sys0_reg
*/
extern adc_mux0_register *adc_mux0_ptr;
extern adc_mux1_register *adc_mux1_ptr;
extern adc_sys0_register *adc_sys0_ptr;

/**
*       \brief Initialize pins for interfacing with ADC
*
*       ADS1247 - MSP430 pin-out:
 *      MCU:    ADC:
 *      P1.5    DRDY*
 *      P5.2    CS*
 *      P5.3    RST*
 *      P9.0    START
 *      P9.1    SCLK // USCI A2
 *      P9.2    DIN  // USCI A2
 *      P9.3    DOUT // USCI A2
*/
void adc_spi_pins_init (void);

/**
 *      \brief Initialize SPI communication with ADC
 */
void adc_spi_init (void);

/**
 *      \brief Reset ADS1247
 *
 *      @return Returns adc_set state
 */
device_state adc_reset (void);
```

```
/**
 *      \brief Read single register from ADS1247
 *
 *      @param *adc_rreg_cmd         The first element of the ADC rreg command
 *      @return adc_register         Register value returned by the ADC
 */
uint8_t adc_spi_rreg_single (uint8_t *adc_rreg_cmd);

void adc_spi_rreg_first4 (uint8_t *adc_rreg_cmd, uint8_t *four_reg_stg);

/**
 *      \brief Read 3 bytes of calibration registers OFC or FSC
 *
 *      @param *cal_reg_cmd          Pointer to the read calibration register command
 *      @param *adc_cal_register     Pointer to the storage for the calibration register
 */
void adc_spi_rreg_cal_register (uint8_t *cal_reg_cmd, uint8_t *adc_cal_register);

/**
 *      \brief Read all registers from ADS1247
 */
void adc_spi_rreg_all (uint8_t *adc_regs_all);

/**
 *      \brief Defines the contents of the write command for the ADC MUX0 register
 *
 *      The first byte defines a type of a write command, the second byte defines
 *      number of register to write, the third byte defines the register value to be
 *      written.
 *
 *      @param  *adc_mux0_param      Pointer to the struct containing MUX0 parameters
 */
void adc_spi_define_wreg_mux0 (adc_mux0_register *adc_mux0_param);

/**
 *      \brief Defines the contents of the write command for the ADC MUX1 register
 *
 *      The first byte defines a type of a write command, the second byte defines
 *      number of register to write, the third byte defines the register value to be
 *      written.
 *
 *      @param  *adc_mux1_param      Pointer to the struct containing MUX1 parameters
 */
void adc_spi_define_wreg_mux1 (adc_mux1_register *adc_mux1_param);

/**
 *      \brief Defines the contents of the write command for the ADC SYS0 register
 *
 *      The first byte defines a type of a write command, the second byte defines
 *      number of registers to write, the third byte defines the register value to be
 *      written.
 *
 *      @param  *adc_mux0_param      Pointer to the struct containing SYS0 parameters
 */
void adc_spi_define_wreg_sys0 (adc_sys0_register *adc_sys0_param);

/**
 *      \brief Write single register (either mux0, mux1 or sys0) to ADS1247
 *
```

```
 *      The function accesses the adc_wreg_single array which is prepopulated by
adc_spi_define_wreg_xxx command,
 *      that is why no parameters are passed to the function.
 *
 *      @return status                Returned status is success or fail
 */
uint8_t adc_spi_wreg_single (void);

/**
 *      \brief Block-write registers mux1 and sys0
 *
 *      When performing multiple individual write commands to the first four registers,
 *      wait at least 64 oscillator clocks before initiating another write command,
 *      which translates into 32 us or 640 MCLK cycles. To alleviate it, the command to make
 *      a block write of mux1 and sys0 is used.
 *
 *      @param *adc_mux1_param       Ptr to the struct containing mux1 register parameters
 *      @param *adc_sys0_param       Ptr to the struct containing sys0 register parameters
 *      @return status               Returned status is success or fail
 */
uint8_t adc_spi_wreg_mux1_sys0 (adc_mux1_register *adc_mux1_param, adc_sys0_register
*adc_sys0_param);

uint8_t adc_spi_wreg_mux0 (void);

/**
 * \brief Set up ADC parameters for measurements
 *
 * @param      data_rate      Value of the sys0 register that corresponds to specific data rate
 * @param      mux1_vrefcon   Value of the vrefcon field in mux1 register; controls VREF module
 * @return     device_state   Return state adc_measuring
 */
device_state adc_setup (uint8_t data_rate, uint8_t mux1_vrefcon);

/**
 *      \brief Receive one conversion result over SPI
 *
 */
void adc_spi_rdata_once (void);

/**
 *      \brief Perform self-offset calibration of ADS1247
 */
void adc_selfocal (void);

/**
 *      \brief Perform system offset calibration
 */
void adc_sysocal (void);

/**
 *      \brief Perform system gain calibration of ADS1247
 *
 *      @param data_rate      Value of the sys0 register that corresponds to specific data rate
 *      @return status        Returned status is success or fail
 */
uint8_t adc_sysgcal (uint8_t data_rate);

/**
 *      \brief Performs the ADC's calibration
 *
```

```
 *      @param data_rate      Value of the sys0 register that corresponds to specific data rate
 *      @return device_state  Returned device_state is adc_set if success or adc_broken if fail
 */
device_state adc_calibration (uint8_t data_rate);

/**
 *      \brief Function that performs ADC measurements for a single channel
 *
 *      @param num_conv       Number of conversions to make
 *      @param *storage       Pointer to storage for the conv results (in RAM)
 *      @param index          Index of the position of the next conv. result in ram storage
 *      @return index
 */
uint16_t adc_measure (uint16_t num_conv, uint8_t *storage, uint16_t index);

uint16_t adc_measure_all_channels (uint8_t *storage, uint16_t index, uint16_t *crc);

#endif /* ADC_DRIVER_H_ */
```

crc8.c

```
/* --COPYRIGHT--,BSD
 * Copyright (c) 2016, Texas Instruments Incorporated
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * *  Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * *  Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * *  Neither the name of Texas Instruments Incorporated nor the names of
 *    its contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
 * OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 * --/COPYRIGHT--*/
//*****************************************************************************
/*
 * crc8.c
 *
 *  Created on: Apr 1, 2017
 *      Author: kdolgikh
 *      Uses the TI driverlib
 */
```

```
#include "stdint.h"
#include "crc8.h"
#include "driverlib.h"
#include "measure.h"
#include "data_prep.h"
#include "real_time_clock.h"

uint16_t crc8_set (uint8_t *input_data_ptr,
                   uint16_t data_length)
{
        uint16_t crc_seed = 0xDEAD;
        uint16_t crc_result;
        uint16_t i;

        CRC_setSeed(CRC_BASE, crc_seed);

        for(i = 0; i < data_length; i++)
        {
                CRC_set8BitData(CRC_BASE,*(input_data_ptr+i)); //Add data into the CRC signature
        }

        crc_result = CRC_getResult(CRC_BASE);

        return crc_result;
}

uint8_t crc8_check (uint16_t crc_result,
                    uint8_t *input_data_ptr,
                    uint16_t data_length)
{
        uint16_t crc_seed = 0xDEAD;
        uint16_t i;

        CRC_setSeed(CRC_BASE, crc_seed);

        for(i = 0; i < data_length; i++)
        {
                CRC_set8BitData(CRC_BASE,*(input_data_ptr+i));
        }

        if(crc_result == CRC_getResult(CRC_BASE))
                return CORRECT_DATA;
        else return BAD_DATA;
}


void check_data_crc(uint8_t *storage, operation_mode op_mode)
{
        uint16_t crc_val = 0;
        uint16_t i = 0;

        switch(op_mode)
        {
                case(rtc_operation):
                        // Check time stamp CRC
                        crc_val = int_merge(storage + TS_CRC_BYTE1_POS);
                        *(storage + TS_CRC_BYTE1_POS) =
crc8_check(crc_val,storage,TS_NUM_BYTES);
                        *(storage + TS_CRC_BYTE2_POS) = 0; // set the second byte of CRC to 0

                        // Check conv results CRC
```

```
                        while (i <= (SEGMENT_SIZE - TS_CRC_CONV_CRC_SP_BYTES))
                        {
                                crc_val = int_merge(storage +TS_CRC_CONV_BYTES +i);

                                *(storage + TS_CRC_CONV_BYTES + i) = crc8_check(crc_val,

                        (storage +TS_W_CRC_NUM_BYTES +i), CONV_LENGTH_16CH);

                                *(storage + TS_CRC_CONV_BYTES + i + 1) = 0; // set the second
byte of CRC to 0
                                i += CONV_CRC_LGTH;
                        }
                        break;
                case(bulk_operation):
                        crc_val = int_merge(storage + NUM_BYTES_SEGM_BULK);
                        *(storage + NUM_BYTES_SEGM_BULK) =
crc8_check(crc_val,storage,NUM_BYTES_SEGM_BULK);
                        *(storage + NUM_BYTES_SEGM_BULK + 1) = 0;
                        break;
        }
}

void replace_ram_stg_crc(uint8_t *storage, uint16_t index)
{
        uint16_t i = TS_CRC_CONV_BYTES;

        // Replace TS CRC with "correct data" token
        *(storage + TS_CRC_BYTE1_POS) = CORRECT_DATA;
        *(storage + TS_CRC_BYTE2_POS) = 0;

        // Replace conv results CRC with "correct data" token
        while (i <= (index - CRC_BYTES))
        {
                *(storage + i) = CORRECT_DATA;
                *(storage + i + 1) = 0;
                i += CONV_CRC_LGTH;
        }
}
```

crc8.h

```
/*
 * crc8.h
 *
 *  Created on: Apr 1, 2017
 *      Author: kdolgikh
 */

#ifndef CRC8_H_
#define CRC8_H_

#include "stdint.h"
#include "adc_driver.h"
#include "states.h"

#define CRC_BYTES     2       // The number of CRC bytes
#define BAD_DATA      0xBD    // BD stands for "bad data"
#define CORRECT_DATA  0xCD    // CD stands for "correct data"

/**
```

```
 * \brief Generates CRC signature
 *
 * @param   input_data_ptr   Input data to generate CRC signatures
 * @param   data_length      Length of the data
 * @return  crc_result       Returns CRC result
 */
uint16_t crc8_set (uint8_t *input_data_ptr, uint16_t data_length);

/**
 * \brief Checks calculated CRC of conversion results against the stored CRC values
 *
 * @param      crc_result     The stored CRC signature
 * @param      input_data_ptr Input data to generate CRCs to compare with stored CRCs
 * @param      data_length    Length of the data
 * @return     status         Returns GOOD_DATA or BAD_DATA for host PC
 */
uint8_t crc8_check (uint16_t crc_result, uint8_t *input_data_ptr, uint16_t data_length);


void check_data_crc(uint8_t *storage, operation_mode op_mode);

// No need to check CRC in RAM since it hasn't been written to flash or SD-card
void replace_ram_stg_crc(uint8_t *storage, uint16_t index);

#endif /* CRC8_H_ */
```

data_prep.c

```
/*
 * data_prep.c
 *
 * Created on: 02/01/2017
 *      Author: kdolgikh
 */
#include "stdint.h"
#include "data_prep.h"

void int_divide (uint16_t data16, uint8_t *data8_ptr)
{
	*data8_ptr = (data16 >> 8) & 0xFF;
	*(data8_ptr + 1) = data16 & 0xFF;
}

uint16_t int_merge (uint8_t *data8_ptr)
{
	uint16_t temp0 = 0;
	uint16_t temp1 = 0;
	uint16_t temp2 = 0;

	temp0 = *data8_ptr;
	temp1 = *(data8_ptr+1);
	temp2 = (temp0 << 8) | temp1;

	return temp2;
}

void long_int_divide (uint32_t *data32_ptr, uint8_t *data8_ptr, uint8_t data32_length)
{
	uint8_t i;
	uint16_t k = 0;
```

```
	for (i=0; i < data32_length; i++)
	{
		*(data8_ptr+k)  =	(*(data32_ptr+i) >> 24) & 0xFF;		//MSB
		*(data8_ptr+k+1) =	(*(data32_ptr+i) >> 16) & 0xFF;
		*(data8_ptr+k+2) =	(*(data32_ptr+i) >> 8) & 0xFF;
		*(data8_ptr+k+3) =	*(data32_ptr+i) & 0xFF;			//LSB
		k+=4;
	}
}

void long_int_merge (uint8_t *data8_ptr, uint32_t *data32_ptr, uint8_t data32_length)
{
	uint8_t i;
	uint16_t k = 0;
	uint32_t temp0 = 0;
	uint32_t temp1 = 0;
	uint32_t temp2 = 0;
	uint32_t temp3 = 0;

	for (i=0; i < data32_length; i++)
	{// temporary variables are used instead of cast operator
		temp0 = *(data8_ptr+ k);	// MSB
		temp1 = *(data8_ptr+k+1);
		temp2 = *(data8_ptr+k+2);
		temp3 = *(data8_ptr+k+3);	// LSB
		*(data32_ptr+i) = (temp0<<24) | (temp1<<16) | (temp2<<8) | temp3;
		k+=4;
	}
}

void reverse_order (uint8_t *data_source_stg, uint8_t *data_dest_stg, uint16_t size)
{
	uint16_t i=0;
	uint16_t j=0;

	while (i < size)
	{
		*(data_dest_stg+i) = *(data_source_stg+j+3);
		*(data_dest_stg+i+1) = *(data_source_stg+j+2);
		*(data_dest_stg+i+2) = *(data_source_stg+j+1);
		*(data_dest_stg+i+3) = *(data_source_stg+j);
		i+=4;
		j+=4;
	}
}
```

data_prep.h

```
/*
 * data_prep.h
 *
 *  Created on: 02/01/2017
 *      Author: kdolgikh
 */

#ifndef DATA_PREP_H_
#define DATA_PREP_H_

#include "stdint.h"
```

```
#define CRC_NUM_BYTES   2   // Number of bytes after CRC value uint16t is divided into two
values uint8_t

/**
 * \brief Divide uint16_t into two uint8_t and store into array pointed by data8_ptr
 *
 * @param      data16         uint16_t variable to be divided
 * @param      *data8_ptr     Pointer to the array for the result
 */
void int_divide (uint16_t data16, uint8_t *data8_ptr);

/**
 * \brief Merge two uint8_t into uint16_t
 *
 * @param      *data8_ptr     Pointer to the array of two uint8_t variables
 * @return                    Return uint16_t variable
 */
uint16_t int_merge (uint8_t *data8_ptr);

/**
* \brief Divides uint32_t array into uint8_t array
*
* @param       *data32_ptr    Ptr to the input array of uint32_t
* @param       *data8_ptr     Ptr to the output array of uint8_t
* @param       data32_length  Length of the uint32_t array
*/
void long_int_divide (uint32_t *data32_ptr, uint8_t *data8_ptr, uint8_t data32_length);

/**
* \brief Gets uint8_t array and merges it into uint32_t array
*
* @param *data8_ptr           Ptr to the input array of uint8_t (source)
* @param *data32_ptr          Ptr to the output array of uint32_t (destination)
* @param data32_length        Length of the uint32_t array
*/
void long_int_merge (uint8_t *data8_ptr, uint32_t *data32_ptr, uint8_t data32_length);

//Reverse order is used because data in flash (and hence SD-card) is uint32
void reverse_order (uint8_t *data_source_stg, uint8_t *data_dest_stg, uint16_t size);

#endif /* DATA_PREP_H_ */
```

hal_SPI.c

```
/* --COPYRIGHT--,BSD
 * Copyright (c) 2016, Texas Instruments Incorporated
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * *  Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * *  Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
```

```
 * *  Neither the name of Texas Instruments Incorporated nor the names of
 *    its contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
 * OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 * --/COPYRIGHT--*/
//*****************************************************************************
/*
 * hal_SPI.c
 *
 *  Created on: Apr 1, 2017
 *      Author: kdolgikh
 *      Uses the TI driverlib
 */

#ifndef _SPILIB_C
#define _SPILIB_C
//
//*****************************************************************
#include "hal_SPI.h"
#include "driverlib.h"

//Send one byte via SPI
uint8_t spiSendByte(uint8_t data)
{
  while (!(UCB1IFG & UCTXIFG));    // wait while not ready for TX
         UCB1TXBUF = data;         // write data
  while (!(UCB1IFG & UCRXIFG));    // wait for RX buffer (full)
         return (UCB1RXBUF);
}

//Read a frame of bytes via SPI
uint8_t spiReadFrame(uint8_t *pBuffer, uint16_t size)
{
       uint16_t i = 0;

       // Receive data
       for (i = 0; i < size; i++)
       {
              while (!(UCB1IFG & UCTXIFG));   // wait while not ready for TX
                     UCB1TXBUF = 0xFF;        // dummy write
              while (!(UCB1IFG & UCRXIFG));   // wait for RX buffer (full)
                     pBuffer[i] = UCB1RXBUF;
       }
       return(0);
}

//Send a frame of bytes via SPI
uint8_t spiSendFrame(uint8_t *pBuffer, uint16_t size)
{
       uint8_t dummy_byte = 0;
```

```
        uint16_t i = 0; // clock the actual data transfer and receive the bytes
        for (i = 0; i < size; i++)
        {
                while (!(UCB1IFG & UCTXIFG));    // wait while not ready for TX
                        UCB1TXBUF = *(pBuffer+i); // write
                while (!(UCB1IFG & UCRXIFG));    // wait for RX buffer (full)
                        dummy_byte = UCB1RXBUF;
        }

        return dummy_byte;
}

//------------------------------------------------------------------------
#endif /* _SPILIB_C */
```

hal_SPI.h

```
#ifndef _SPILIB_H
#define _SPILIB_H

#include "stdint.h"

uint8_t spiSendByte(uint8_t data);

uint8_t spiReadFrame(uint8_t *pBuffer, uint16_t size);

uint8_t spiSendFrame(uint8_t *pBuffer, uint16_t size);

#endif /* _SPILIB_H */
```

measure.c

```
/*
 * measure.c
 *
 *  Created on: Dec 19, 2017
 *      Author: kdolgikh
 */

#include "driverlib.h"
#include "sleep_timer.h"
#include "adc_driver.h"
#include "sdcard.h"
#include "crc8.h"
#include "data_prep.h"
#include "mux_adg706_driver.h"
#include "measure.h"
#include "real_time_clock.h"
#include "send_over_usb.h"

uint8_t num_segm_bulk = NUM_SEGMENTS_BULK;
uint32_t meas_rate = 0;
uint16_t crc_value = 0;
uint8_t *flash_data8_ptr = (uint8_t *)BANK1_START;
uint32_t *flash_data32_ptr = (uint32_t *)BANK1_START;
uint16_t index = TS_W_CRC_NUM_BYTES;
uint8_t ram_stg[SEGMENT_SIZE] = {0}; // RAM storage for conversion results of all channels
```

```
uint32_t ram_stg_32[RAM_STG_32_LGTH] = {0}; // RAM storage for conv results in uint32_t format
uint16_t segment_count = 0;// Counts number of segments to be written to flash
uint32_t num_segments = 0; // Counts total number of segments written to both flash and SDcard
uint8_t sdc_full = FALSE;  // Flag indicating that SD-card is full
uint8_t mem_full = FALSE;  // Flag indicating that device memory is full
uint32_t num_segments2sdc = NUM_SEGMENTS_FLASH;
extern uint32_t max_segments_sd;
uint8_t firstMeasurement = FALSE;
uint8_t newSegment = TRUE;
uint8_t KeepData = FALSE;
uint8_t CalEn = FALSE;
exit_point usb_exit;

device_state measure_bulk_all_channels_store_flash (uint8_t num_segm)
{
	uint8_t i;
	uint8_t k = 0;
	uint16_t crc_value = 0;

    flash_data8_ptr = (uint8_t *)BANK1_START; // Reinit flash ptr to start writing to Bank 1
    flash_data32_ptr = (uint32_t *)BANK1_START;

	while (k < 16)
	{
		switch_mux_ch(k+1);

		for (i = 0; i < num_segm; i++)
		{
			adcWorkInProgress = TRUE;

			adc_measure(NUM_MEAS_SEGM_BULK,ram_stg,0);// 170 conversions to fill the
segment of 512 byte

			adcWorkInProgress = FALSE;

			crc_value = crc8_set(ram_stg,NUM_BYTES_SEGM_BULK); // Generate the CRC
signature of a 170 results block

			int_divide(crc_value,(ram_stg+SEGMENT_SIZE-CRC_BYTES)); // Divide CRC
value into uint8t and place into 510, 511 element of ram_stg

			write_data2flash();

			if(USBconnected)
			{
				usb_exit = exit4;
				return usb_connected;
			}
		}
		k++;

		LPM3; // Exit from LPM by pressing the button after switching resistor

		if(USBconnected)
		{
			usb_exit = exit5;
			return usb_connected;
		}
	}
	return adc_measuring;
}
```

```
device_state measure_all_channels_store_flash_sdc (void)
{
    while (1) //repeat until device memory is full
    {
        while (segment_count < NUM_SEGMENTS_FLASH) // repeat until flash is full
        {
              while (index < SEGMENT_SIZE - NUM_SPARE_BYTES) // repeat until RAM storage is
full, then write to flash.
            {
              // Checks cases of USB connection:
              // 1) USB was connected during measurement when RAM stg didn't become full
              // 2) USB was connected during flash write when flash didn't become full
              // 3) USB was connected during SD-card write
              if(!USBconnected)
                     {
                            if(firstMeasurement)
                                   firstMeasurement = FALSE;
                            else
                            {
                                   LPM3;

                                   if(USBconnected) // If the device was woken up by USB
connection and not meas interval expiration
                                          {
                                                 usb_exit = exit5;
                                                 return usb_connected;
                                          }
                                   }

                     if(newSegment)
                     {
                            gen_time_stamp(ram_stg); // insert time stamp into ram_stg for
the first measurement in each segment
                            newSegment = FALSE;
                     }
                     }
              else
              {
                     usb_exit=exit4;
                     return usb_connected;
              }

              if (CalEn)
              {
                     adc_calibration(adc_data_rate);
                     adc_setup(adc_data_rate, adc_mux1_vrefcon);
              }

              index = adc_measure_all_channels(ram_stg, index, &crc_value);
            }

            newSegment = TRUE; // Every segment should have a time stamp

            // Checks cases of USB connection:
            // 1) USB was connected during the last conversion of the segment (the code didn't
enter while(index..) loop
            if(USBconnected)
            {
              usb_exit=exit6;
              return usb_connected;
```

```
            }

            index = TS_W_CRC_NUM_BYTES; // Reinit index for the next loop run

            // If connected to USB before the segment was written to flash, it will be
retrieved directly from RAM,
            // so no need to write the last segment to flash and account it as being written
to flash and later SD-card
            segment_count++; // this value is reinit every time it count up to
NUM_SEGMENTS_FLASH
            num_segments++; // this value shows total num of segments written to both flash
and SD-card

            write_data2flash();

            if (num_segments == max_segments_sd)                  // Last segment that can be
written to SD-card until it's full
            {
                  num_segments2sdc = (uint32_t)(flash_data8_ptr - BANK1_START) >> 9; //
Divide by 512 (or right shift by 9) to get number of segments to write to SD-card.
                  break; // Break while (segment_count ...) loop
            }
         }

        segment_count = 0;
       // Reinit segment count for the next while(segment_count ...) run

        if (sdc_full)
            break;  // break while(1) loop // Do not write to SD-card if it's full

        // Checks cases of USB connection:
        // 1) USB was connected during flash write when it became full (the code didn't enter
while(segment_count...) loop)
        // Program exits with end address of flash Bank 3, which indicates that flash has been
filled with data,
        // or with flash address where it has stopped due to max_segments_sd condition.
        if(USBconnected)
        {
              usb_exit=exit7;
              return usb_connected;
        }

        flash_data8_ptr = (uint8_t *)BANK1_START; // Reinit flash ptr to start writing to SD-
card and flash from Bank 1
        flash_data32_ptr = (uint32_t *)BANK1_START; // Reinit flash ptr to start writing to
flash in the next loop from Bank 1

        write_data2sdc(sdc_block_address, flash_data8_ptr, num_segments2sdc);

        if (num_segments != max_segments_sd)
              sdc_block_address += (uint32_t)SDC_ADDR_INCREMENT; // Points to the block to
start writing next time, in bytes
        else
        {
              sdc_block_address += (num_segments2sdc << 9); // The last addr points to the end
of SD-card, which is used in send_conv_result
              sdc_full = TRUE;
        }
    }

    RTCPS1CTL &= ~RT1PSIE; // disable interrupt to turn off measure intervals
```

```
    // Once done testing, turn of RTC clock here as well as you will not need it

    mem_full = TRUE;

    if(USBconnected)
    {
       usb_exit=exit8;
       return usb_connected;
    }
    else
       return dev_mem_full;
}

void write_data2flash (void)
{
    uint8_t status = 0;

    long_int_merge(ram_stg,ram_stg_32,RAM_STG_32_LGTH); // Merge values in RAM storage into
uint32_t variables
    // Writing uint32t is 2x faster than writing uint8_t/uint16_t
    // write RAM storage data to flash
     do
     {
         FlashCtl_eraseSegment(flash_data8_ptr);
         status = FlashCtl_performEraseCheck(flash_data8_ptr,SEGMENT_SIZE); // Check that
segment contains only 0xFF
     }
     while(status == STATUS_FAIL);
     FlashCtl_write32(ram_stg_32,flash_data32_ptr,RAM_STG_32_LGTH);

     flash_data8_ptr += SEGMENT_SIZE; // Increase flash8 pointer to point to next segment
     flash_data32_ptr += RAM_STG_32_LGTH; // Increase flash32 pointer to point to next segment
}

void write_data2sdc (uint32_t block_addr, uint8_t *data_ptr, uint32_t num_segments)
{

    sdcPowerOn();

    if (sdcStart() == SDC_SUCCESS) // Set SD-card in SPI-mode
        sdc_SPI_Init();            // Increase clock frequency to 20 MHz

    sdc_response = sdcWriteMultipleBlocks(block_addr,     // Start from this block
                                          data_ptr,       // Write this data
                                          num_segments);  // Write this number of segments
    sdcPowerOff();
}

void mem_counters_cleanup(void)
{
       index = TS_W_CRC_NUM_BYTES;
       segment_count = 0;
       num_segments = 0;
    flash_data8_ptr = (uint8_t *)BANK1_START;
    flash_data32_ptr = (uint32_t *)BANK1_START;
       num_segments2sdc = NUM_SEGMENTS_FLASH;
       sdc_block_address = 0;
}
```

measure.h

```
/*
 * measure.h
 *
 *  Created on: Dec 19, 2017
 *      Author: kdolgikh
 */

#ifndef MEASURE_H_
#define MEASURE_H_

#include "stdint.h"
#include "states.h"
#include "crc8.h"
#include "real_time_clock.h"
#include "adc_driver.h"

/// \name Memory related definitions
//@{
#define BANK1_START          0x28000     // Start address of Bank 1
#define BANK2_START          0x48000     // Start address of Bank 2
#define BANK3_START          0x68000     // Start address of Bank 3
#define SEGMENT_SIZE         512         // Main flash memory segment size, byte
#define NUM_SEGMENTS_FLASH   768         // Number of 512 Byte segments in 384 KB flash
#define SDC_ADDR_INCREMENT   NUM_SEGMENTS_FLASH*SEGMENT_SIZE // Address increment when writing
flash to sdc
#define RAM_STG_32_LGTH      (SEGMENT_SIZE>>2) // Ram storage contains 128 uint32t values
#define NUM_MEAS_SEGM_BULK   170     // Number of measurements per segment in bulk mode
#define NUM_BYTES_SEGM_BULK  510     // Number of meas bytes in the segment in bulk mode
#define NUM_SEGMENTS_BULK    30      // Number of bulk conversion segments of 512 bytes, should
be not more than 48 if writing only to flash
#define FLASH_BULK_BYTES     NUM_CHANNELS*NUM_SEGMENTS_BULK*SEGMENT_SIZE
//@}


#define NUM_CONV             10      // Number of times to measure all channels
#define SINGLE_CONV          1       // Single conversion
#define NUM_SPARE_BYTES      5       // Number of spare bytes in RAM storage of SEGMENT_SIZE
                                     // left after writing TS (5 Bytes) with CRC (2 Bytes)
                                     // and 10 measurement results for 16 channels with CRC
(500 Bytes).

#define CONV_LENGTH_16CH     48      // Length of conversion results for 16 channels

// Misc cumulative length of different field in storage segment of 512 bytes
#define CONV_CRC_LGTH        (CONV_LENGTH_16CH +CRC_BYTES) // 50
#define TS_CRC_CONV_BYTES    (TS_NUM_BYTES +CRC_BYTES +CONV_LENGTH_16CH) // 55
#define TS_CRC_CONV_CRC_SP_BYTES (TS_CRC_CONV_BYTES +CRC_BYTES +NUM_SPARE_BYTES) // 62

extern uint8_t firstMeasurement;
extern uint32_t meas_rate;
extern exit_point usb_exit;

/**
 *  Number of segments to use for bulk conversion. Should be not more than 48
 */
extern uint8_t num_segm_bulk;
```

```
/**
 * Array in RAM to hold conversion results for all channels
 */
extern uint8_t ram_stg [SEGMENT_SIZE];

/**
 * RAM storage for conv results in uint32_t format
 */
extern uint32_t ram_stg_32[RAM_STG_32_LGTH];
extern uint16_t index;
extern uint8_t *flash_data8_ptr; // Pointer used to erase flash segments and for SD-card write
extern uint32_t *flash_data32_ptr; // Ptr used to write 32 bit conv results with CRC to flash

extern uint8_t sdc_full;
extern uint8_t mem_full;
extern uint32_t num_segments2sdc;
extern uint8_t KeepData;
extern uint8_t CalEn;

device_state adc_measure_single_channel(uint8_t channel_num,
                                        uint32_t num_conv);

// Measure channels in bulk and store to flash.
device_state measure_bulk_all_channels_store_flash (uint8_t num_segm);

// The function's organization allows writing entire flash once SD-card is full
device_state measure_all_channels_store_flash_sdc (void);

void write_data2flash (void);

void write_data2sdc (uint32_t block_addr, uint8_t *data_ptr, uint32_t num_segments);

void mem_counters_cleanup (void);

/*device_state measure (uint8_t channel_num,
                      uint32_t num_conv);*/

#endif /* MEASURE_H_ */
```

mux_adg706_driver.c

```
/*
 * mux_adg706_driver.c
 *
 *  Created on: Dec 2, 2016
 *      Author: kdolgikh
 */

/** \file mux_adg706_driver.c
 *     \brief Analog Devices 16 channel mux ADG706B driver source file
 */

#ifndef MUX_ADG706_DRIVER_C_
#define MUX_ADG706_DRIVER_C_

#include "msp430.h"
#include "mux_adg706_driver.h"
#include "states.h"
```

```
void mux_adg706_ports_init (void)
{
       P4DIR |= A3+A2+A1+A0+MUX_EN;     // Set output direction
       P4OUT &= ~(A3+A2+A1+A0+MUX_EN); // Set output LOW; MUX is disabled.
}

void disable_mux_adg706 (void)
{
       P4OUT &= ~MUX_EN;
}

void switch_mux_ch (uint8_t ch_num)
{
    switch(ch_num)
    {
        case(CHANNEL1):
            switch_mux_adg706_ch1();               // Switch to channel 1
            break;
        case(CHANNEL2):
            switch_mux_adg706_ch2();
            break;
        case(CHANNEL3):
            switch_mux_adg706_ch3();
            break;
        case(CHANNEL4):
            switch_mux_adg706_ch4();
            break;
        case(CHANNEL5):
            switch_mux_adg706_ch5();
            break;
        case(CHANNEL6):
            switch_mux_adg706_ch6();
            break;
        case(CHANNEL7):
            switch_mux_adg706_ch7();
            break;
        case(CHANNEL8):
            switch_mux_adg706_ch8();
            break;
        case(CHANNEL9):
            switch_mux_adg706_ch9();
            break;
        case(CHANNEL10):
            switch_mux_adg706_ch10();
            break;
        case(CHANNEL11):
            switch_mux_adg706_ch11();
            break;
        case(CHANNEL12):
            switch_mux_adg706_ch12();
            break;
        case(CHANNEL13):
            switch_mux_adg706_ch13();
            break;
        case(CHANNEL14):
            switch_mux_adg706_ch14();
            break;
        case(CHANNEL15):
            switch_mux_adg706_ch15();
            break;
        case(CHANNEL16):
```

```
            switch_mux_adg706_ch16();           // Switch to channel 16
            break;
    }
}

void switch_mux_adg706_ch1 (void)
{
       P4OUT &= ~(A3+A2+A1+A0);
       P4OUT |= MUX_EN;
}

void switch_mux_adg706_ch2 (void)
{
       P4OUT &= ~(A3+A2+A1);
       P4OUT |= A0+MUX_EN;
}

void switch_mux_adg706_ch3 (void)
{
       P4OUT &= ~(A3+A2+A0);
       P4OUT |= A1+MUX_EN;
}

void switch_mux_adg706_ch4 (void)
{
       P4OUT &= ~(A3+A2);
       P4OUT |= A1+A0+MUX_EN;
}

void switch_mux_adg706_ch5 (void)
{
       P4OUT &= ~(A3+A1+A0);
       P4OUT |= A2+MUX_EN;
}

void switch_mux_adg706_ch6 (void)
{
       P4OUT &= ~(A3+A1);
       P4OUT |= A2+A0+MUX_EN;
}

void switch_mux_adg706_ch7 (void)
{
       P4OUT &= ~(A3+A0);
       P4OUT |= A2+A1+MUX_EN;
}

void switch_mux_adg706_ch8 (void)
{
       P4OUT &= ~A3;
       P4OUT |= A2+A1+A0+MUX_EN;
}

void switch_mux_adg706_ch9 (void)
{
       P4OUT &= ~(A2+A1+A0);
       P4OUT |= A3+MUX_EN;
}
```

```
void switch_mux_adg706_ch10 (void)
{
	P4OUT &= ~(A2+A1);
	P4OUT |= A3+A0+MUX_EN;
}

void switch_mux_adg706_ch11 (void)
{
	P4OUT &= ~(A2+A0);
	P4OUT |= A3+A1+MUX_EN;
}

void switch_mux_adg706_ch12 (void)
{
	P4OUT &= ~A2;
	P4OUT |= A3+A1+A0+MUX_EN;
}

void switch_mux_adg706_ch13 (void)
{
	P4OUT &= ~(A1+A0);
	P4OUT |= A3+A2+MUX_EN;
}

void switch_mux_adg706_ch14 (void)
{
	P4OUT &= ~A1;
	P4OUT |= A3+A2+A0+MUX_EN;
}

void switch_mux_adg706_ch15 (void)
{
	P4OUT &= ~A0;
	P4OUT |= A3+A2+A1+MUX_EN;
}

void switch_mux_adg706_ch16 (void)
{
	P4OUT |= A3+A2+A1+A0+MUX_EN;
}

#endif /* MUX_ADG706_DRIVER_C_ */
```

mux_adg706_driver.h

```
/*
 * mux_adg706_driver.h
 *
 *  Created on: Dec 2, 2016
 *      Author: kdolgikh
 */

/** \file mux_adg706_driver.h
 *      \brief Analog Devices 16 channel mux ADG706B driver header file
 */

#ifndef MUX_ADG706_DRIVER_H_
#define MUX_ADG706_DRIVER_H_
```

```c
#include "msp430.h"
#include "stdint.h"
#include "states.h"

#define A0 BIT0
#define A1 BIT1
#define A2 BIT2
#define A3 BIT3
#define MUX_EN BIT4

#define ALL_CHANNELS    0
#define CHANNEL1        1
#define CHANNEL2        2
#define CHANNEL3        3
#define CHANNEL4        4
#define CHANNEL5        5
#define CHANNEL6        6
#define CHANNEL7        7
#define CHANNEL8        8
#define CHANNEL9        9
#define CHANNEL10       10
#define CHANNEL11       11
#define CHANNEL12       12
#define CHANNEL13       13
#define CHANNEL14       14
#define CHANNEL15       15
#define CHANNEL16       16

/**     \brief Set up MUX ADG706 ports
 *
 *      Ports connections:
 *  MCU         MUX
 *      P4.0    A0
 *      P4.1    A1
 *      P4.2    A2
 *      P4.3    A3
 *      P4.4    EN
 */
void mux_adg706_ports_init (void);

/** \brief Disable MUX ADG706
 *
 */
void disable_mux_adg706 (void);

/**
 *      \brief Switch MUX ADG706 to the specific channel_number
 */
void switch_mux_ch (uint8_t channel_number);

/** \brief Switch MUX ADG706 channels
 *
 */
void switch_mux_adg706_ch1 (void);
void switch_mux_adg706_ch2 (void);
void switch_mux_adg706_ch3 (void);
void switch_mux_adg706_ch4 (void);
void switch_mux_adg706_ch5 (void);
void switch_mux_adg706_ch6 (void);
void switch_mux_adg706_ch7 (void);
void switch_mux_adg706_ch8 (void);
```

```
void switch_mux_adg706_ch9 (void);
void switch_mux_adg706_ch10 (void);
void switch_mux_adg706_ch11 (void);
void switch_mux_adg706_ch12 (void);
void switch_mux_adg706_ch13 (void);
void switch_mux_adg706_ch14 (void);
void switch_mux_adg706_ch15 (void);
void switch_mux_adg706_ch16 (void);

#endif /* MUX_ADG706_DRIVER_H_ */
```

real_time_clock.c

```
/*
 * real_time_clock.c
 *
 *  Created on: Dec 28, 2017
 *      Author: kdolgikh
 */
#include "driverlib.h"
#include "real_time_clock.h"
#include "data_prep.h"
#include "crc8.h"
#include "measure.h"
#include "sleep_timer.h"

uint8_t rtc_sec = 0;  // rw
uint8_t rtc_min = 0;  // rw
uint8_t rtc_hour = 0; // rw
uint8_t rtc_day = 0;  // Day of month, rw
uint8_t rtc_mon = 0;  // Month, rw
uint16_t rtc_year = 0;       // rw

uint32_t time_stamp = 0;
volatile uint32_t rt1ps_count = 0;   // counts num of RT1IP interrupts
uint8_t start_immediately = FALSE;   // start immediately after disconnection
uint8_t start_zero = FALSE;          // start at 00 clock (00 sec, xx min 00 sec, xx hours 00
min 00 sec, etc)

uint16_t ts_crc = 0;  // time stamp CRC
uint8_t ts_debug_array[7]={0};

void gen_time_stamp (uint8_t *ts_storage_ptr)
{
       uint8_t i;

       while (!(RTCCTL1 & 0x10))// If RTCRDY (0x10) is LOW (registers are being updated), wait
              ;

       rtc_year = RTCYEAR; // Retrieve TS data. Could do it inside RTC ISR
       rtc_mon = RTCMON;
       rtc_day = RTCDAY;
       rtc_hour = RTCHOUR;
       rtc_min = RTCMIN;
       rtc_sec = RTCSEC;

       // Year will be in the format 18-255 for 2018-2255 accordingly (to meet uint8_t)
       // Store Year in ts_storage
       *(ts_storage_ptr) = (uint8_t)rtc_year;
```

```
        time_stamp = ((uint32_t)rtc_mon)<<28            // Generate MDHMS time stamp
                                | ((uint32_t)rtc_day)<<23
                                | ((uint32_t)rtc_hour)<<18
                                | ((uint32_t)rtc_min)<<12
                                | ((uint32_t)rtc_sec)<<6; // 6 Bits are unused

        // Divide time_stamp into 4 uint8_t and store in ts_storage
        long_int_divide(&time_stamp, (ts_storage_ptr+TS_BYTES_POSITION), 1);

        // Generate CRC for Year + MDHMS
        ts_crc = crc8_set(ts_storage_ptr,TS_NUM_BYTES);

        // Divide CRC into 2 uint8_t and store in ts_storage
        int_divide(ts_crc,(ts_storage_ptr + TS_CRC_BYTE1_POS));


        for (i=0;i<7;i++)
                *(ts_debug_array+i) = *(ts_storage_ptr+i);
}
```

real_time_clock.h

```
/*
 * real_time_clock.h
 *
 *  Created on: Dec 28, 2017
 *      Author: kdolgikh
 */

#ifndef REAL_TIME_CLOCK_H_
#define REAL_TIME_CLOCK_H_

#include "stdint.h"

#define TS_BYTES_POSITION 1 // Position in RAM stg to start writing 4 Bytes of M-D-H-Min-Sec
#define TS_CRC_BYTE1_POS  5 // Position in RAM storage to start writing 2 Bytes of TS CRC
#define TS_CRC_BYTE2_POS  6 // Position of the second TS CRC byte
#define TS_YEAR_LGTH      1 // Number of bytes in Year
#define TS_NUM_BYTES      5 // Number of bytes of TS (Year + MDHMS)
#define TS_W_CRC_NUM_BYTES   (TS_NUM_BYTES +CRC_BYTES) // 7

extern uint8_t rtc_sec;
extern uint8_t rtc_min;
extern uint8_t rtc_hour;
extern uint8_t rtc_day;
extern uint8_t rtc_mon;
extern uint16_t rtc_year;
extern uint32_t time_stamp;
extern volatile uint32_t rt1ps_count;
extern uint8_t start_immediately;
extern uint8_t start_zero;

void gen_time_stamp(uint8_t *ts_storage_ptr);

#endif /* REAL_TIME_CLOCK_H_ */
```

sdcard.c

```
/*
 * sdcard.c
 *
 *  Created on: Apr 15, 2017
 *      Author: kdolgikh
 */

#ifndef _SDCARD_C
#define _SDCARD_C
//
//-------------------------------------------------------------

#include "hal_SPI.h"
#include "sdcard.h"
#include "sleep_timer.h"

uint8_t sdcInitialized_event = FALSE;
//uint8_t sdc_buffer[SDC_RW_BLOCK_SIZE] = {0};
uint8_t sdc_register_buff[SDC_CSD_CID_REG_LENGTH] = {0};
uint8_t sdc_cmd_frame[SDC_CMD_LGTH] = {0};
uint32_t sdc_block_address = 0; // address is in bytes

// Calculate this value for each card separately
// by using values from CSD register, see page 35, 109
uint16_t sdc_read_access_time = READ_ACCESS_DELAY;

uint16_t sdcSize = SDC_128MB;

uint8_t sdc_response = 0;

uint8_t r1_response = 0;            // R1 response or the first byte of R2 response
uint8_t r2_response = 0;            // The second byte of R2 response
uint8_t data_resp_token = 0;
uint8_t data_response = 0;
uint8_t busy_response = 0;

uint16_t nac = 0;

uint32_t num_segments_sdc (uint16_t sdCardSize)
{
    uint32_t num_segments;  // number of 512 byte segments on the SD card of specific size if
using raw write (without File System)

    switch(sdCardSize)
    {
        case(SDC_128MB):
            num_segments = 248320; //3; //262144; //248320 from WinHex
               break;
        case(SDC_256MB):
            num_segments = 524288; // need to check
               break;
        case(SDC_512MB):
            num_segments = 1048576; // need to check
               break;
        case(SDC_1024MB):
            num_segments = 2097152; // need to check
               break;
        case(SDC_2048MB):
            num_segments = 4194304; // need to check
```

```
            break;
    }
    return num_segments;
}

void sdc_CS_Init(void)
{
    // Configure Chip Select
    SDC_CS_PxOUT &= ~SDC_CS; // CS LOW -> OK after I unmounted R11
                             // PxOUT is undefined after reset, hence configure before PxDIR
    SDC_CS_PxDIR |= SDC_CS;  // Set CS DIR to output
}

void sdc_SPI_Init(void)
{
    UCB1CTL1 |= UCSWRST;            // Hold USCI in reset

    if (!sdcInitialized_event)
        {
        UCB1CTL0 |= UCMST;          // 3-pin, 8-bit master mode
        UCB1CTL0 |= UCCKPL;         // Clock polarity
        UCB1CTL0 |= UCMSB;          // MSB first
        UCB1CTL0 |= UCSYNC;         // SPI mode
        UCB1CTL1 |= UCSSEL__SMCLK;  // BRCLK = SMCLK (20MHz)
        UCB1BR0 = 0x40;             // Divide BRCLK by 64 to get SCLK of 312.5 KHz
                                    // During sd-card init SCLK should be > 400KHz
        UCB1BR1 = 0;                // Set to 0 to use only UCB1BR0 value

        SDC_PxSEL |= SDC_SIMO + SDC_SOMI + SDC_UCLK; // Activate USCI functionality on port 8
        }
    else
        UCB1BR0 = 0x00;             // 0 for SCLK of 20 MHz, 10 for SCLK of 2 MHz

    UCB1CTL1 &= ~UCSWRST;           // Start USCI state machine
}

void sdcDeactivatePorts(void)
{
    UCB1CTL1 |= UCSWRST;                            // Hold USCI in reset
    SDC_PxSEL &= ~(SDC_SIMO + SDC_SOMI + SDC_UCLK); // Deactivate USCI functionality on port 8
    SDC_PxOUT &= ~(SDC_SIMO + SDC_SOMI + SDC_UCLK); // Set LOW
    SDC_PxDIR |= SDC_SIMO + SDC_SOMI + SDC_UCLK;    // Set output direction
}

void sdcPowerOn (void)
{
    SDC_CS_PxOUT |= SDC_CS;         // Set CS HIGH
    P4OUT |= BIT7;                  // turn SD_VCC on
    sdc_SPI_Init();
    Sleep_Timer_Fast(40);           // Wait 10 ms until sd-card initializes
}

void sdcPowerOff (void)
{
    sdcDeactivatePorts();
    P4OUT &= ~BIT7;              // turn SD_VCC off
    SDC_CS_PxOUT &= ~SDC_CS;     // Set CS LOW
    sdcInitialized_event = FALSE;// SD-card will wake up in uninitialized state after power up
}
```

```
// send command to SD
void sdcSendCmd (const uint8_t cmd, uint32_t data, const uint8_t crc)
{
	sdc_cmd_frame[0]=(cmd | SDC_CMD_MSB);
	sdc_cmd_frame[1] = (data >> 24) & 0xFF;		//MSB
	sdc_cmd_frame[2] = (data >> 16) & 0xFF;
	sdc_cmd_frame[3] = (data >> 8) & 0xFF;
	sdc_cmd_frame[4] = data & 0xFF;				//LSB
	sdc_cmd_frame[5] = crc;

	spiSendFrame(sdc_cmd_frame,6);
}

// sdc Get R1 or R2 Response (when sending CMD13)
void sdcGet_R1_R2_Response(uint8_t r2ResponseRequired)
{
	//Response comes 1-8 bytes after command
	//data will be 0xFF until response

	uint8_t i = 0;
	uint8_t break_flag = FALSE;

	while((i <= N_CR_MAX) && (!break_flag)) // Ncr is 1 to 8 bytes, byte 9 - response,
hence <= sign
	{
		r1_response = spiSendByte(SDC_NOP);

		if (r1_response != SDC_NOP) // It is the task of overlaying function to check
that response is valid
			break_flag = TRUE;
		i++;
	}

	if (r2ResponseRequired)
		r2_response = spiSendByte(SDC_NOP);

	spiSendByte(SDC_NOP); // 8 clock cycles are required after the response
				// See page 53 SD physical layer spec. v2.00, p 96 SanDisk UG
}


// Initialize SD card in SPI mode
uint8_t sdcStart (void)
{
	uint8_t i;

	//initialization sequence on PowerUp

	// Send > 74 SCLK ticks (in this case 8 bits/byte * 10 bytes = 80 bits)
	for(i = 0; i < SDC_INIT_BYTES ; i++)
		spiSendByte(SDC_NOP);

	// Set CS Low
	SDC_CS_PxOUT &= ~SDC_CS;

	//Send Command 0 to put SD in SPI mode
	sdcSendCmd(SDC_GO_IDLE_STATE, SDC_ZERO_ARGUMENT, SDC_CMD1_CONST_CRC);

	//Response should be 0x01
	sdcGet_R1_R2_Response(FALSE);
```

```
        // CS High
        SDC_CS_PxOUT |= SDC_CS;

        if(r1_response != SDC_R1_IDLE_STATE)
                return SDC_INIT_ERROR;

        //Wait until card finishes initializing and sends 0x00 (as the only possible response
for this command)
        while(r1_response == SDC_R1_IDLE_STATE)
        {
                // Increase delay, if necessary
                spiSendByte(SDC_NOP); // 8 clock cycles are used as delay between CS transitions
                                      // SD-card starts counting clocks after CS is asserted Low

                SDC_CS_PxOUT &= ~SDC_CS; // CS Low

                sdcSendCmd(SDC_SEND_OP_COND, SDC_ZERO_ARGUMENT, SDC_NOP); // External delay is
not required here, because // function prepares cmd for some time
                sdcGet_R1_R2_Response(FALSE);

                SDC_CS_PxOUT |= SDC_CS;
        }

        sdcInitialized_event = TRUE; // This flag is used to increase SCLK after initialization

        return SDC_SUCCESS;
}


// Set block_length
// 1 < blocklength < 512
void sdcSetBlockLength (const uint16_t blocklength)
{
        SDC_CS_PxOUT &= ~SDC_CS;

        spiSendByte(SDC_NOP); // Ncs

        // Set the block length
        sdcSendCmd(SDC_SET_BLOCKLEN, blocklength, SDC_NOP);

        // Receive R1 response
        sdcGet_R1_R2_Response(FALSE);

        SDC_CS_PxOUT |= SDC_CS;
}


// Check if SD-card is busy
void sdcCheckBusy(void)
{
        // Wait until SD-card is not busy
        do
        {
                busy_response = spiSendByte(SDC_NOP);
        }
        while(busy_response == SDC_BUSY); // SD-card holds SOMI Low when busy

        spiSendByte(SDC_NOP); //Nwr (page 109 SanDisk) or Nec (page 108 SanDisk)
}
```

```
uint8_t sdcGetDataResponse(void)
{
	data_resp_token = spiSendByte(SDC_NOP);
	data_resp_token &= 0x1F;
	return data_resp_token;
}

void sdcStopTransmission (void)
{
		spiSendByte(SDC_NOP); // Ncr, see p108 SanDisk "Card Response to Host Command"
				      // Additional time between two commands

		// Send Stop Transmission command
		sdcSendCmd(SDC_STOP_TRANSMISSION, SDC_ZERO_ARGUMENT, SDC_NOP);

		// Receive R1 response
		sdcGet_R1_R2_Response(FALSE);
}


// read a size "block_size" block beginning at the "address"
// 1 < block_size < 512
// 1 < buffer_size < 512
uint8_t sdcReadSingleBlock(const uint32_t address, const uint16_t block_size, uint8_t
*pBuffer)
{
	uint16_t i = 0;

	// Set the block length
	sdcSetBlockLength(block_size);
	if (r1_response == SDC_R1_SUCCESS)   // block length can be set
	{
		SDC_CS_PxOUT &= ~SDC_CS;

		spiSendByte(SDC_NOP); // Ncs

		sdcSendCmd(SDC_READ_SINGLE_BLOCK,address, SDC_NOP); // send read command
SDC_READ_SINGLE_BLOCK=CMD17
		sdcGet_R1_R2_Response(FALSE); // receive R1 response

		if (r1_response == SDC_R1_SUCCESS)
		{
			// Receive data start token
			// Data token for read command will be sent by SD-card after Nac
			// Nac_max = 100*(TAAC*fclk + NSAC*100), see page 109, page 35
			// Where TAAC and NSAC are values from CSD register
			while(i < sdc_read_access_time)
			{
				data_resp_token = spiSendByte(SDC_NOP);
				if(data_resp_token == SDC_START_SINGLE_BLOCK_READ)
					break;
				i++;
				nac++;
			}

			//If desired response was not received during sdc_access_time, return
SDC_RDATA_ERROR
			if (data_resp_token != SDC_START_SINGLE_BLOCK_READ)
			{
				SDC_CS_PxOUT |= SDC_CS;
				return SDC_RDATA_ERROR;
```

```
                        }

                        // receive data
                        spiReadFrame(pBuffer, block_size);

                        // get 2 CRC bytes
                        spiSendByte(SDC_NOP);
                        spiSendByte(SDC_NOP);

                        // required 8 clock cycles after data block
                        spiSendByte(SDC_NOP);

                        SDC_CS_PxOUT |= SDC_CS;

                        return SDC_SUCCESS;
                }
                else
                {
                        SDC_CS_PxOUT |= SDC_CS;
                        return SDC_RCMD_ERROR;
                }
        }
        else
                return SDC_BLOCK_SET_ERROR;
}

uint8_t sdcWriteSingleBlock (const uint32_t address, uint8_t *pBuffer)
{
        // Set the block length to write, block size should be equal 512 bytes
        sdcSetBlockLength(SDC_RW_BLOCK_SIZE);

        if (r1_response == SDC_R1_SUCCESS)   // block length could be set
        {
                SDC_CS_PxOUT &= ~SDC_CS;

                spiSendByte(SDC_NOP); // Ncs

                // send write command
                sdcSendCmd(SDC_WRITE_BLOCK, address, SDC_NOP);

                sdcGet_R1_R2_Response(FALSE);

                if (r1_response == SDC_R1_SUCCESS)
                {
                        // Send the start token
                        spiSendByte(SDC_START_SINGLE_BLOCK_WRITE);

                        // Send data
                        spiSendFrame(pBuffer, SDC_RW_BLOCK_SIZE);

                        // Send 2 CRC bytes
                        spiSendByte(SDC_NOP);
                        spiSendByte(SDC_NOP);

                        // Get data response token
                        data_response = sdcGetDataResponse();

                        if (data_response == SDC_R_TOKEN_DATA_ACCEPTED)
                                sdcCheckBusy();       // Check busy

                        // Check write status, see page 94 SanDisk
```

```
                        spiSendByte(SDC_NOP); // Ncr of 1 byte between two commands, see page
108 SanDisk
                        sdcSendCmd(SDC_SEND_STATUS, SDC_ZERO_ARGUMENT, SDC_NOP);  // Send Status
command
                        sdcGet_R1_R2_Response(TRUE); // Get r1 and r2 responses

                        SDC_CS_PxOUT |= SDC_CS;

                        if (r1_response == SDC_R1_SUCCESS)
                        {
                                if (r2_response == SDC_R2_SUCCESS)
                                        return SDC_SUCCESS;
                                else
                                        return SDC_WDATA_ERROR;
                        }
                        else
                                return SDC_SEND_STATUS_ERROR;
                }
                else
                {
                        SDC_CS_PxOUT |= SDC_CS;
                        return SDC_WCMD_ERROR;
                }
        }
        else
                return SDC_BLOCK_SET_ERROR;
}

uint8_t sdcWriteMultipleBlocks (const uint32_t address, uint8_t *pBuffer, uint32_t count)
{
        uint32_t i;

        // Set the block length to write, block size should be equal 512 bytes
        sdcSetBlockLength(SDC_RW_BLOCK_SIZE);

        if (r1_response == SDC_R1_SUCCESS)   // block length can be set
        {
                SDC_CS_PxOUT &= ~SDC_CS;

                spiSendByte(SDC_NOP); // Ncs

                // send write command
                sdcSendCmd(SDC_WRITE_MULTIPLE_BLOCK, address, SDC_NOP);

                sdcGet_R1_R2_Response(FALSE);

                if (r1_response == SDC_R1_SUCCESS)
                {
                        for (i = 0; i < count; i++)
                        {
                                // Send the start token
                                spiSendByte(SDC_START_MULTIPLE_BLOCK_WRITE);

                                // Send data
                                spiSendFrame(pBuffer, SDC_RW_BLOCK_SIZE);

                                pBuffer +=SDC_RW_BLOCK_SIZE;

                                // Send 2 CRC bytes
                                spiSendByte(SDC_NOP);
                                spiSendByte(SDC_NOP);
```

```
                        // Get data response token
                        data_response = sdcGetDataResponse();

                        if (data_response == SDC_R_TOKEN_DATA_ACCEPTED)
                                sdcCheckBusy();       // Check busy // Ideally, after busy
should check results of the programming (CMD13), see page 94 SanDisk
                        else
                        {
                                sdcStopTransmission(); // stop transmission if error is
encountered
                                // In case of write error, additional actions are to send
CMD13 and ACMD22 (see page 104 SanDisk)

                                if (data_response == SDC_R_TOKEN_DATA_RJCT_WRITE_ERR)
                                {
                                        sdcSendCmd(SDC_SEND_STATUS, SDC_ZERO_ARGUMENT,
SDC_NOP); // Send Status command
                                        sdcGet_R1_R2_Response(TRUE); // Get r1 and r2
responses

                                        if (r1_response != SDC_R1_SUCCESS)
                                        {
                                                SDC_CS_PxOUT |= SDC_CS;
                                                return SDC_SEND_STATUS_ERROR;
                                        }

                        // It's the task for overlaying function to check R2 response
                        // Checking the num of well written blocks (ACMD22) is not implemented
                                }

                                SDC_CS_PxOUT |= SDC_CS;

                                return data_response;
                        }
                }
                // Send the stop token once all data is sent
                spiSendByte(SDC_STOP_MULTIPLE_BLOCK_WRITE);
                spiSendByte(SDC_NOP);         // Nbr, see page 109 SanDisk
                sdcCheckBusy();                       // After Nbr card may send Busy

                SDC_CS_PxOUT |= SDC_CS;

                return SDC_SUCCESS;
        }
        else
        {
                SDC_CS_PxOUT |= SDC_CS;
                return SDC_WCMD_ERROR;
        }
    }
    else
        return SDC_BLOCK_SET_ERROR;
}

// Reading the contents of the CSD and CID registers in SPI mode is a simple read-block
transaction.
// CID register length is 16 byte (128 bit), page 32
// CSD register length is 16 byte (128 bit), page 33
// When issuing CSD read, read access time is unknown, use Ncr instead, see page 95
// Issue CID read after CSD read, so read access time is known
```

```
uint8_t sdcReadRegister (const uint8_t cmd_register, uint8_t *pBuffer)
{
	// Ideally, length of the buffer is needed to be checked

	uint16_t i = 0;

	sdcSetBlockLength(SDC_CSD_CID_REG_LENGTH);

	if (r1_response == SDC_R1_SUCCESS)
	{
		SDC_CS_PxOUT &= ~SDC_CS;

		spiSendByte(SDC_NOP); // Ncs

		sdcSendCmd(cmd_register, SDC_ZERO_ARGUMENT, SDC_NOP);

		sdcGet_R1_R2_Response(FALSE);

		if (r1_response == SDC_R1_SUCCESS)
		{
			// Receive data start token
			// Data token for read CSD/CID command will be sent by SD-card after Ncx
			// Ncx_max = 8 Bytes, see page 109
			// However, the value of read access time will be used
			while(i < sdc_read_access_time)
			{
				data_resp_token = spiSendByte(SDC_NOP);
				if(data_resp_token == SDC_START_SINGLE_BLOCK_READ)
					break;
				i++;
			}

	//If desired response was not received during sdc_access_time, return SDC_RDATA_ERROR
			if (data_resp_token != SDC_START_SINGLE_BLOCK_READ)
				return SDC_RDATA_ERROR;

			// Receive data
			spiReadFrame(pBuffer, SDC_CSD_CID_REG_LENGTH);

			// Get 2 CRC bytes
			spiSendByte(SDC_NOP);
			spiSendByte(SDC_NOP);

			// Required 8 clock cycles after data block
			spiSendByte(SDC_NOP);

			SDC_CS_PxOUT |= SDC_CS;

			return SDC_SUCCESS;
		}
		else
		{
			SDC_CS_PxOUT |= SDC_CS;
			return SDC_RCMD_ERROR;
		}
	}
	else
		return SDC_BLOCK_SET_ERROR;
}
//------------------------------------------------------------------------
#endif /* _SDCARD_C */
```

sdcard.h

```
/*
 * sdcard.h
 *
 *  Created on: Apr 15, 2017
 *      Author: kdolgikh
 */

#ifndef _SDCARD_H
#define _SDCARD_H

#include "stdint.h"
#include "msp430f5659.h"

#define TRUE                                                    1
#define FALSE                                                   0

// SPI port definitions
#define SDC_PxSEL                                               P8SEL
#define SDC_PxDIR                                               P8DIR
#define SDC_PxIN                                                P8IN
#define SDC_PxOUT                                               P8OUT
#define SDC_SIMO                                                BIT5
#define SDC_SOMI                                                BIT6
#define SDC_UCLK                                                BIT4

// Chip Select
#define SDC_CS_PxDIR                                            P8DIR
#define SDC_CS_PxOUT                                            P8OUT
#define SDC_CS                                                  BIT7

// commands related defines
#define SDC_INIT_BYTES        10      // Number of bytes sent to SD-card during initialization
#define SDC_NOP               0xFF    // Dummy send / CRC don't care
#define SDC_ZERO_ARGUMENT     0       // If a command doesn't require argument, it should be 0
#define SDC_CMD_MSB           0X40    // MSB of any sdc command is 0x40
#define SDC_CMD1_CONST_CRC    0x95    // CMD1 CRC plus the last bit of the command yields 0x95
#define N_CR_MAX              8       // Max command response time Ncr, Byte
#define N_CX_MAX              8       // Max response time to read CSD command, Byte
#define SDC_CMD_LGTH          6       // Length of the command, Byte
#define SDC_RW_BLOCK_SIZE     512     // SD-card block size (read/write unit)
#define SDC_CSD_CID_REG_LENGTH 16     // Length of the CSD and CID registers, Byte

// SD-card responses:
// R1 response to any command except SEND_STATUS, 1 Byte
#define SDC_R1_SUCCESS        0x00
#define SDC_R1_IDLE_STATE     0x01    // The card is in idle state and running initializing
process
#define SDC_R1_ERASE_RESET    0x02    // Erase sequence was cleared before executing because
ERASE_SEQ_ERR was received
#define SDC_R1_ILLEGAL_CMD    0x04
#define SDC_R1_COM_CRC_ERR    0x08
#define SDC_R1_ERASE_SEQ_ERR 0x10     // An error in the sequence of erase commands
#define SDC_R1_ADDR_ERROR     0x20    // Misaligned address
#define SDC_R1_PARAM_ERROR    0x40    // Command's argument (address, block length) was out of
the allowed range for this card

// R2 response to SEND_STATUS, 2 Bytes: Byte 1 (MSB) is identical to R1, Byte 2 (LSB) below:
#define SDC_R2_SUCCESS        0x00
#define SDC_R2_CARD_LOCKED    0x01
```

```
#define SDC_R2_WP_ERASE_SKIP 0x02    // Also, Lock/Unlock Failed
#define SDC_R2_ERROR         0x04    // General or unknown error
#define SDC_R2_CC_ERROR      0x08    // Card Controller error
#define SDC_R2_CARD_ECC_FAILED 0x10
#define SDC_R2_WP_VIOLATION  0x20
#define SDC_R2_ERASE_PARAM   0x40    // Invalid sector for erase

// R3 response to READ_OCR, 5 Bytes: Byte 1 - R1, Bytes 2-5 - OCR

// Data write response token, page 104
// xxx00101 & 0x1F
// xxx01011 & 0x1F
// xxx01101 & 0x1F
#define SDC_R_TOKEN_DATA_ACCEPTED    0x05    // data to be written is accepted
#define SDC_R_TOKEN_DATA_RJCT_CRC    0x0B    // data to be written is rejected due to CRC error
#define SDC_R_TOKEN_DATA_RJCT_WRITE_ERR 0x0D       // data to be written is rejected due to
write error

// Data read/write tokens, page 104
#define SDC_START_SINGLE_BLOCK_READ     0xFE   // Data token start byte. Send by SD-card
#define SDC_START_MULTIPLE_BLOCK_READ  0xFE   // Data token start byte. Send by SD-card
#define SDC_START_SINGLE_BLOCK_WRITE   0xFE   // Data token start byte. Send by host
#define SDC_START_MULTIPLE_BLOCK_WRITE 0xFC   // Data token start byte. Send by host
#define SDC_STOP_MULTIPLE_BLOCK_WRITE  0xFD   // Data token stop byte. Send by host

#define READ_ACCESS_DELAY 1000 // Delay (Bytes) between the last bit of the read cmd and the
first bit of data

// Data read token, page 105
#define SDC_DATA_ERR_TOKEN_ERROR            0x01
#define SDC_DATA_ERR_TOKEN_CC_ERROR         0x02
#define SDC_DATA_ERR_TOKEN_CARD_ECC_FAILED  0x04
#define SDC_DATA_ERR_OUT_OF_RANGE           0x08
#define SDC_BUSY                            0x00   // SD-card holds SOMI Low while busy

// Status Register (different from SD Status Register)
// Only 16 bits containing errors relevant to SPI can be read out of 32 bits SR

// error/success statuses
#define SDC_SUCCESS           0x00  // any SD-card operation is successful
#define SDC_BLOCK_SET_ERROR   0x02  // command to set block length returned an error
#define SDC_WCMD_ERROR        0x03  // write (W) command returned an error
#define SDC_WDATA_ERROR       0x04  // write (W) operation failed
#define SDC_RCMD_ERROR        0x05  // read (R) command returned an error
#define SDC_RDATA_ERROR       0x06  // read operation failed
#define SDC_INIT_ERROR        0x07  // initialization failed
#define SDC_SEND_STATUS_ERROR 0x08  // Error during Send Status command

// CMD-number mnemonic in HEX, see pages 99-102 SanDisk SD Card
#define SDC_GO_IDLE_STATE           0x00   //CMD0
#define SDC_SEND_OP_COND            0x01   //CMD1
#define SDC_READ_CSD                0x09   //CMD9
#define SDC_READ_CID                0x0A   //CMD10
#define SDC_STOP_TRANSMISSION       0x0C   //CMD12
#define SDC_SEND_STATUS             0x0D   //CMD13
#define SDC_SET_BLOCKLEN            0x10   //CMD16
#define SDC_READ_SINGLE_BLOCK       0x11   //CMD17
#define SDC_READ_MULTIPLE_BLOCK     0x12   //CMD18
#define SDC_WRITE_BLOCK             0x18   //CMD24
#define SDC_WRITE_MULTIPLE_BLOCK    0x19   //CMD25
#define SDC_WRITE_CSD               0x1B   //CMD27
```

```
#define SDC_SET_WRITE_PROT          0x1C   //CMD28
#define SDC_CLR_WRITE_PROT          0x1D   //CMD29
#define SDC_SEND_WRITE_PROT         0x1E   //CMD30
#define SDC_ERASE_WR_BLK_START_ADDR 0x20   //CMD32
#define SDC_ERASE_WR_BLK_END_ADDR   0x21   //CMD33
#define SDC_EREASE                  0x26   //CMD38
#define SDC_LOCK_UNLOCK             0x2A   //CMD42
#define SDC_APP_CMD                 0x37   //CMD55
#define SDC_READ_OCR                0x3A   //CMD58
#define SDC_CRC_ON_OFF              0x3B   //CMD59

#define SDC_128MB   128                   // SD standard capacity card sizes
#define SDC_256MB   256
#define SDC_512MB   512
#define SDC_1024MB  1024
#define SDC_2048MB  2048

extern uint16_t sdc_read_access_time;
extern uint8_t sdcInitialized_event;
extern uint8_t sdc_register_buff[SDC_CSD_CID_REG_LENGTH];
extern uint8_t sdc_cmd_frame[SDC_CMD_LGTH];
extern uint32_t sdc_block_address; // Address of the block on SD-card in bytes
extern uint16_t sdcSize;           // Size of SD-card. User enters during setup

extern uint8_t sdc_response;       // Response from SD-card returned by SD-card functions

extern uint8_t r1_response;        // R1 response or the first byte of R2 response
extern uint8_t r2_response;        // The second byte of R2 response
extern uint8_t data_resp_token;
extern uint8_t data_response;
extern uint8_t busy_response;

uint32_t num_segments_sdc (uint16_t sdCardSize);

// initialize CS pin for SD-card
void sdc_CS_Init(void);

// Initialize SPI for SD-card
void sdc_SPI_Init(void);

// When SD-card is powered off, pull CS and data lines HIGH,
// pull clock LOW to prevent floating inputs
void sdcDeactivatePorts(void);

void sdcPowerOn (void);

void sdcPowerOff (void);

// send command to SDC
void sdcSendCmd (const uint8_t cmd, uint32_t data, const uint8_t crc);

void sdcGet_R1_R2_Response(uint8_t r2RespRequired);
// Put SD-card into SPI mode
uint8_t sdcStart (void);

// set SDC block length of count=2^n Byte
void sdcSetBlockLength (const uint16_t blocklength);

void sdcCheckBusy(void);

void sdcStopTransmission (void);
```

```
// read a size Byte big block beginning at the address.
uint8_t sdcReadSingleBlock(const uint32_t address, const uint16_t count, uint8_t *pBuffer);
//#define sdcReadSector(sector, pBuffer) sdcReadSingleBlock(sector*512ul, SD_RW_BLOCK_SIZE,
pBuffer)

// write a 512 Byte big block beginning at the (aligned) address
uint8_t sdcWriteSingleBlock (const uint32_t address, uint8_t *pBuffer);
//#define sdcWriteSector(sector, pBuffer) sdcWriteSingleBlock(sector*512ul, SD_RW_BLOCK_SIZE,
pBuffer)

uint8_t sdcWriteMultipleBlocks (const uint32_t address, uint8_t *pBuffer, uint32_t count);

// Read Register arg1 with Length arg2 (into the buffer)
uint8_t sdcReadRegister (const uint8_t cmd_register, uint8_t *pBuffer);

#endif /* _SDCARD_H */
```

send_over_usb.c

```
/* --COPYRIGHT--,BSD
 * Copyright (c) 2016, Texas Instruments Incorporated
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * *  Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * *  Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * *  Neither the name of Texas Instruments Incorporated nor the names of
 *    its contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
 * OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 * --/COPYRIGHT--*/
/*
 * send_over_usb.c
 *
 *  Created on: Mar 31, 2017
 *      Author: kdolgikh
 *      Uses TI USB library
 */
```

```
#include "stdint.h"
#include "driverlib.h"
#include "USB_config/descriptors.h"
#include "USB_API/USB_Common/device.h"
#include "USB_API/USB_Common/usb.h"      // USB-specific functions
#include "USB_API/USB_CDC_API/UsbCdc.h"
#include "USB_app/usbConstructs.h"
#include "send_over_usb.h"
#include "data_prep.h"
#include "crc8.h"
#include "states.h"
#include "adc_driver.h"
#include "sleep_timer.h"
#include "measure.h"
#include "sdcard.h"
#include "real_time_clock.h"
#include "ucs_functions.h"

extern device_state dev_state;
extern uint8_t dev_state_old;
extern operation_mode op_mode;
extern volatile uint8_t DRDY_event;
volatile uint8_t USB_DataReceived_event = FALSE; // Flag showing the reception of the message
from PC

uint8_t usb_msg_trx[USB_MSG_TRX_LGTH] = {0};  // Array for the message to/from PC
uint8_t NumDataBytes_Sent = FALSE;

// for debug, move inside functions once done debugging
uint32_t total_bytes = 0;
uint32_t flash_bytes = 0;
uint32_t sdc_address = 0;
uint8_t *flash_address = (uint8_t *)BANK1_START;
uint8_t sdc_fail_count = 0;

extern uint16_t nac; // for debug only

uint8_t ram_stg_temp[SEGMENT_SIZE] = {0}; // for data from RAM, the size will be "index"
uint8_t ram_stg_temp2[SEGMENT_SIZE] = {0}; // for data from sd-card

uint8_t USBconnected = FALSE;

void usb_connect2host (void)
{
       if (USB_enable() == USB_SUCCEED) // Connect to host
       {
              USB_reset();
              USB_connect();
       }

       LPM0; // Device should be in AM or LPM0 during enumeration.
            // Wait here until device is enumerated.
}
void usb_command_line (void)
{
    USBCDC_receiveDataInBuffer(usb_msg_trx,USB_MSG_TRX_LGTH,CDC0_INTFNUM);
       switch(*usb_msg_trx)  // the first byte of the message is the command code
       {
              case(GET_CONV_RESULTS):
                     usb_send_conv_result();
                     break;
```

```
                case(USB_GET_NUM_SEGM_BULK):
                case(USB_GET_ADC_DATA_RATE):
                case(USB_GET_ADC_VREFCON):
                case(USB_GET_DEV_STATE):
                case(USB_GET_SYSOCAL_EN):
                case(USB_GET_DELAY_EN):
                case(USB_GET_KEEP_DATA):
                case(USB_GET_CAL_EN):
                case(USB_GET_OP_MODE):
                        usb_get_dev_param();
                        break;
                case(USB_GET_MEAS_RATE):
                        usb_get_meas_rate();
                        break;
                case(USB_SET_NUM_SEGM_BULK):
                case(USB_SET_ADC_DATA_RATE):
                case(USB_SET_DEV_STATE):
                case(USB_SET_SYSOCAL_EN):
                case(USB_SET_DELAY_EN):
                case(USB_SET_MEAS_RATE):
                case(USB_SET_MEAS_START):
                case(USB_SET_KEEP_DATA):
                case(USB_SET_CAL_EN):
                case(USB_SET_OP_MODE):
                        usb_set_dev_param();
                        break;
                case(USB_SET_ADC_VREFCON):
                        usb_set_adc_vrefcon();
                        break;
                case(USB_GET_ADC_OFC):
                case(USB_GET_ADC_FSC):
                        usb_get_adc_cal_register();
                        break;
                case(USB_GET_RTC):
                        usb_get_rtc();
                        break;
                case(USB_SET_RTC):
                        usb_set_rtc();
                        break;
                default:
                        break;
        }
}

void usb_ack(void)
{
        uint16_t num_tx_bytes = 0; // number of TX bytes over USB, returned by abortSend

        // acknowledge success of the set command by sending the received cmd
        if(USBCDC_sendDataAndWaitTillDone(usb_msg_trx,ACK_BUFF_LGTH,CDC0_INTFNUM,
                                          USB_TX_RETRY_NUM))
                USBCDC_abortSend(&num_tx_bytes, CDC0_INTFNUM);
}
void usb_send_conv_result (void)
{
        uint16_t num_tx_bytes = 0;
        uint16_t i;

        if (!NumDataBytes_Sent)
        {
                // Let the host know on the number of bytes to receive
```

```
                flash_bytes = (uint32_t)(flash_data8_ptr - BANK1_START); // need to make sure
that flash_data8_ptr will hold what I need!!!

                if (index > TS_W_CRC_NUM_BYTES) // Data was written to ram_stg
                        total_bytes = sdc_block_address + flash_bytes + (uint32_t)index;
                else // Data was not written to ram_stg
                        total_bytes = sdc_block_address + flash_bytes;

                long_int_divide(&total_bytes,(usb_msg_trx+PARAM_POS),1);  // 1 is number of
uint32_t containing in total_bytes

                if(USBCDC_sendDataAndWaitTillDone(usb_msg_trx, // send ACK plus num_bytes
                                                  NUM_BYTE_BUFF_LGTH, CDC0_INTFNUM,
                                                  USB_TX_RETRY_NUM))
                        USBCDC_abortSend(&num_tx_bytes, CDC0_INTFNUM);

                NumDataBytes_Sent = TRUE;
        }
        else
        {
                NumDataBytes_Sent = FALSE;

// Send ACK to the host // It is easier to send ACK like this than embedding it with data
                usb_ack();

                if(USBconnected)
                {
                        if (index > TS_W_CRC_NUM_BYTES) // Move data from ram_stg into
ram_stg_temp since it will be sent the last
                        {
                                for (i = 0; i < index; i++)
                                {
                                        *(ram_stg_temp+i) = *(ram_stg+i);
                                }
                        }

                        if (sdc_block_address && USBconnected) // Data was written to SD-card
                        {
                                sdcPowerOn();

                                if (sdcStart() == SDC_SUCCESS) // Set SD-card in SPI-mode
                                        sdc_SPI_Init();

                                while ((sdc_address < sdc_block_address) && USBconnected)
                                {
                                        Do // Sometimes SD-card returns 0x06 and 0x02 errors,
power cycling helps mitigating this
                                        {
                                                sdc_response = sdcReadSingleBlock(sdc_address,
SDC_RW_BLOCK_SIZE, ram_stg_temp2);

                                                if(sdc_response)
                                                {
                                                        sdcPowerOff();
                                                        sdc_fail_count++;
                                                        sdcPowerOn();
                                                        if (sdcStart() == SDC_SUCCESS)
// Set SD-card in SPI-mode
                                                                sdc_SPI_Init();
                                                }
                                        }
```

```
                                        while (sdc_response && (sdc_fail_count < 4)); // Power
cycle 3 times only

                                        nac = 0; // for debug only
                                        sdc_fail_count = 0;

                                        sdc_address += SDC_RW_BLOCK_SIZE;

                                        reverse_order(ram_stg_temp2,ram_stg,SEGMENT_SIZE);
//Reverse order is used because data in SD-card is moved from flash where it is stored in
uint32
                                        check_data_crc(ram_stg, op_mode);

                                        if(USBCDC_sendDataAndWaitTillDone(ram_stg,SEGMENT_SIZE,
                                                                          CDC0_INTFNUM,
                                                                          USB_TX_RETRY_NUM))
                                                USBCDC_abortSend(&num_tx_bytes, CDC0_INTFNUM);
                                }
                                sdc_address = 0;      // Reinit sdc address
                                sdcPowerOff();
                        }

                        if (flash_bytes && USBconnected)     // Data has been written to flash
                        {
                                while ((flash_address < flash_data8_ptr) && USBconnected)
                                {
                                        reverse_order(flash_address,ram_stg,SEGMENT_SIZE); //copy
data from flash into ram_stg. Reverse order is used because data in flash is uint32

                                        check_data_crc(ram_stg, op_mode);

                                        if(USBCDC_sendDataAndWaitTillDone(ram_stg,SEGMENT_SIZE,
                                                                          CDC0_INTFNUM,
                                                                          USB_TX_RETRY_NUM))
                                                USBCDC_abortSend(&num_tx_bytes, CDC0_INTFNUM);

                                        flash_address += SEGMENT_SIZE;
                                }
                                flash_address = (uint8_t *)BANK1_START; // Reinit flash address
                        }

                        if ((index > TS_W_CRC_NUM_BYTES) && USBconnected)
                        {
                                replace_ram_stg_crc(ram_stg_temp, index); // replace CRC bytes
with "CD" values

                                if(USBCDC_sendDataAndWaitTillDone(ram_stg_temp,index,
                                                                  CDC0_INTFNUM,
                                                                  USB_TX_RETRY_NUM))
                                        USBCDC_abortSend(&num_tx_bytes, CDC0_INTFNUM);
                        }
                }
        }
}
void usb_get_dev_param (void)
{
        uint16_t num_tx_bytes = 0; // number of TXed bytes over USB, returned by abortSend

        switch(*usb_msg_trx)
        {
        case(USB_GET_NUM_SEGM_BULK):
```

```
                *(usb_msg_trx + PARAM_POS) = num_segm_bulk;
                break;
        case(USB_GET_ADC_DATA_RATE):
                *(usb_msg_trx + PARAM_POS) = adc_data_rate;
                break;
        case(USB_GET_ADC_VREFCON):
                *(usb_msg_trx + PARAM_POS) = adc_mux1_vrefcon;
                break;
        case(USB_GET_DEV_STATE):
                *(usb_msg_trx + PARAM_POS) = dev_state_old;
                break;
        case(USB_GET_SYSOCAL_EN):
                *(usb_msg_trx + PARAM_POS) = sysocal_enable;
                break;
        case(USB_GET_DELAY_EN):
                *(usb_msg_trx + PARAM_POS) = delay_enable;
                break;
        case(USB_GET_KEEP_DATA):
                *(usb_msg_trx + PARAM_POS) = KeepData;
                break;
        case(USB_GET_CAL_EN):
                *(usb_msg_trx + PARAM_POS) = CalEn;
                break;
        case(USB_GET_OP_MODE):
                *(usb_msg_trx + PARAM_POS) = (uint8_t)op_mode;
                break;
        }

        if(USBCDC_sendDataAndWaitTillDone(usb_msg_trx, PARAM_BUFF_LGTH,
                                          CDC0_INTFNUM, USB_TX_RETRY_NUM))
                USBCDC_abortSend(&num_tx_bytes, CDC0_INTFNUM);
}

void usb_get_meas_rate(void)
{
        uint16_t num_tx_bytes = 0; // number of TXed bytes over USB, returned by abortSend

        long_int_divide(&meas_rate,(usb_msg_trx+PARAM_POS),1); // 1 is the number of uint32
values to convert - only meas_rate

        if(USBCDC_sendDataAndWaitTillDone(usb_msg_trx, MEAS_RATE_BUFF_LGTH,
                                          CDC0_INTFNUM, USB_TX_RETRY_NUM))
                USBCDC_abortSend(&num_tx_bytes, CDC0_INTFNUM);
}

void usb_get_adc_cal_register(void)
{
        uint16_t num_tx_bytes = 0;

        uint8_t *cal_reg_cmd_ptr; // pointer to the command that reads a calibration register

        if (*usb_msg_trx == USB_GET_ADC_OFC)
                cal_reg_cmd_ptr = adc_rreg_ofc;
        else if (*usb_msg_trx == USB_GET_ADC_FSC)
                cal_reg_cmd_ptr = adc_rreg_fsc;

        adcWorkInProgress = TRUE;

        P9OUT |= ADC_START;

        adc_spi_rreg_cal_register(cal_reg_cmd_ptr, (usb_msg_trx + PARAM_POS));
```

```
        P9OUT &= ~ADC_START;

        LPM0;

        adcWorkInProgress = FALSE;

        if(USBCDC_sendDataAndWaitTillDone(usb_msg_trx,
                                                        CAL_REG_BUFF_LGTH,
                                                        CDC0_INTFNUM,
                                                        USB_TX_RETRY_NUM))
                USBCDC_abortSend(&num_tx_bytes, CDC0_INTFNUM);
}


void usb_get_rtc(void)
{
        uint16_t num_tx_bytes = 0;

        RTCCTL0 &= ~RTCRDYIFG;       // Clear RTCRDY IFG
        RTCCTL0 |= RTCRDYIE;  // Enable RTCRDY interrupt
        LPM0;  // Sleep until RTCRDY is HIGH

        *(usb_msg_trx + RTC_YEAR_POS)=(uint8_t)RTCYEAR;
        *(usb_msg_trx + RTC_MON_POS)=RTCMON;
        *(usb_msg_trx + RTC_DAY_POS)=RTCDAY;
        *(usb_msg_trx + RTC_HOUR_POS)=RTCHOUR;
        *(usb_msg_trx + RTC_MIN_POS)=RTCMIN;
        *(usb_msg_trx + RTC_SEC_POS)=RTCSEC;

        if(USBCDC_sendDataAndWaitTillDone(usb_msg_trx,USB_RTC_BUFF_LGTH,
                                          CDC0_INTFNUM,USB_TX_RETRY_NUM))
                USBCDC_abortSend(&num_tx_bytes, CDC0_INTFNUM);
}

void usb_set_dev_param(void)
{
        switch(*usb_msg_trx)
        {
        case(USB_SET_NUM_SEGM_BULK):
                num_segm_bulk = *(usb_msg_trx+PARAM_POS);
                break;
        case(USB_SET_ADC_DATA_RATE):
                adc_data_rate = *(usb_msg_trx+PARAM_POS);
                break;
        case(USB_SET_DEV_STATE):
                dev_state_old = *(usb_msg_trx + PARAM_POS); // Device will return to this state
after disconnecting from PC
                break;
        case(USB_SET_SYSOCAL_EN):
                sysocal_enable = *(usb_msg_trx+PARAM_POS);
                break;
        case(USB_SET_DELAY_EN):
                delay_enable = *(usb_msg_trx+PARAM_POS);
                break;
        case(USB_SET_MEAS_RATE):
                long_int_merge(usb_msg_trx+PARAM_POS,&meas_rate,1); // 1 is the number of uint32
values to convert - only meas_rate
                break;
        case(USB_SET_MEAS_START):
                start_immediately = *(usb_msg_trx+PARAM_POS);
```

```
                start_zero = *(usb_msg_trx+PARAM_POS+1);
                break;
        case(USB_SET_KEEP_DATA):
                KeepData = *(usb_msg_trx+PARAM_POS);
                break;
        case(USB_SET_CAL_EN):
                CalEn = *(usb_msg_trx+PARAM_POS);
                break;
        case(USB_SET_OP_MODE):
                op_mode = (operation_mode)(*(usb_msg_trx+PARAM_POS));
                break;
        }

        usb_ack();
}

void usb_set_adc_vrefcon (void)
{
        uint8_t status = 0;
        adc_mux1_vrefcon = *(usb_msg_trx+PARAM_POS);

        // Since this parameter is not set through adc_driver functions, set it in this
function.
        // Valid until next power cycle or reset
        adc_mux1_register adc_mux_1 =
        {
                adc_mux_1.adc_clkstat = MUX1_CLKSTAT_INT_OSC,
                adc_mux_1.adc_vrefcon = *(usb_msg_trx+PARAM_POS),
                adc_mux_1.adc_refselt = MUX1_REFSELT_INT_REF0,
                adc_mux_1.adc_muxcal = MUX1_MUXCAL_GAIN_CAL
        };

        adc_mux1_ptr = &adc_mux_1;

        adc_spi_define_wreg_mux1(adc_mux1_ptr);

        adcWorkInProgress = TRUE;

        P9OUT |= ADC_START;

        do
        {
                status = adc_spi_wreg_single();
        }
        while (status == STATUS_FAIL);

        P9OUT &= ~ADC_START;

        LPM0;

        adcWorkInProgress = FALSE;

        usb_ack();
}
void usb_set_rtc(void)
{
        // See page 600 msp430 UG for writing to RTC when it runs:
        // Takes effect immediately, RTC stops, RT1PS resets

        RTCCTL1 |= 0x40;      // If yes, disable it to configure new clock value
```

```
        RTCYEAR = (uint16_t)(*(usb_msg_trx+RTC_YEAR_POS)); // uint16_t, use cast
        RTCMON = *(usb_msg_trx+RTC_MON_POS);
        RTCDAY = *(usb_msg_trx+RTC_DAY_POS);
        RTCHOUR = *(usb_msg_trx+RTC_HOUR_POS);
        RTCMIN = *(usb_msg_trx+RTC_MIN_POS);
        RTCSEC = *(usb_msg_trx+RTC_SEC_POS);

        RTCCTL1 &= ~0x40; // Release RTC for operation

        usb_ack();
}
```

send_over_usb.h

```
/*
 * send_over_usb.h
 *
 *  Created on: Mar 31, 2017
 *      Author: kdolgikh
 */

#ifndef SEND_OVER_USB_H_
#define SEND_OVER_USB_H_

// the first byte of all messages is always an ACK,
// hence all buffer lengths have an increase by 1 from what the data inside needs

#define USB_TX_RETRY_NUM     100                  // Number of USB TX retries
#define USB_MSG_TRX_LGTH     USB_RTC_BUFF_LGTH    // Length of the USB control message
to/from PC, byte. Max num of bytes that can be sent/received.
#define PARAM_POS            1 // Position of the parameter in the received message. Position
0 is reserved for ACK
#define ACK_BUFF_LGTH        1 // Length of the buffer for ACK
#define PARAM_BUFF_LGTH      2 // Length of the buffer for parameters
#define CAL_REG_BUFF_LGTH    4 // Length of the buffer for calibration register
#define NUM_BYTE_BUFF_LGTH   5 // Length of the buffer for num_byte value
#define MEAS_RATE_BUFF_LGTH  5 // Length of the buffer for meas rate
#define USB_RTC_BUFF_LGTH    7 // Length of the USB msg to/from PC containing RTC data, byte
#define RTC_YEAR_POS         1 // Position of Year in the received message
#define RTC_MON_POS          2 // Position of Month
#define RTC_DAY_POS          3 // Position of Day of Month
#define RTC_HOUR_POS         4 // Position of Hour
#define RTC_MIN_POS          5 // Position of Minute
#define RTC_SEC_POS          6 // Position of Seconds

/**
 * USB communication commands
 */
#define GET_CONV_RESULTS            0x7F   // Transfer conversion results over USB
#define USB_GET_ADC_DATA_RATE       0x03   // Get ADC data rate
#define USB_SET_ADC_DATA_RATE       0x05   // Set ADC data rate
#define USB_GET_NUM_SEGM_BULK       0x07   // Get number of segments containing 170
conversions
#define USB_SET_NUM_SEGM_BULK       0x09   // Set number of segments containing 170
conversions
#define USB_GET_ADC_VREFCON         0x0A   // Get ADC VREFCON value
#define USB_SET_ADC_VREFCON         0x0B   // Set ADC VREFCON value
#define USB_GET_ADC_OFC             0x0C   // Get ADC offset calibration register OFC
#define USB_GET_ADC_FSC             0x0D   // Get ADC gain calibration register FSC
#define USB_GET_DEV_STATE           0x0E   // Get the state of the device
```

```
#define USB_SET_DEV_STATE           0x0F   // Set the state of the device
#define USB_GET_RTC                 0x11   // Get time over USB
#define USB_SET_RTC                 0x12   // Get time over USB
#define USB_GET_SYSOCAL_EN          0x13   // Enable/disable sysocal
#define USB_SET_SYSOCAL_EN          0x14   // Check if sysocal enabled or not
#define USB_GET_DELAY_EN            0x15   // Enable delay before measuring
#define USB_SET_DELAY_EN            0x16   // Check if delay is enabled
#define USB_SET_MEAS_RATE           0x17   // Set measurement rate
#define USB_SET_MEAS_START          0x18   // Set measurement start options
#define USB_GET_MEAS_RATE           0x19   // Get meas rate
#define USB_SET_KEEP_DATA           0x1A   // Set KeepData flag
#define USB_GET_KEEP_DATA           0x1B   // Get KeepData flag
#define USB_SET_CAL_EN              0x1C   // Get Calibration Enabled flag
#define USB_GET_CAL_EN              0x1D   // Set Calibration Enabled flag
#define USB_GET_OP_MODE             0x1E   // Get operational mode
#define USB_SET_OP_MODE             0x1F   // Set operational mode

#include "states.h"

/**
 * \brief Flag showing the reception of the message from PC
 */
extern volatile uint8_t USB_DataReceived_event;

extern uint8_t USBconnected;

void usb_connect2host (void);

/**
 * \brief Command line interface for the device through USB
 */
void usb_command_line (void);

void usb_ack (void);

/**
 * \brief Send the conversion result that is stored in Flash over USB
 *
 */
void usb_send_conv_result (void);

void usb_send_bulk_conv_result (void);

void usb_get_dev_param (void);

void usb_get_meas_rate(void);

void usb_get_adc_cal_register(void);

void usb_get_rtc(void);

void usb_set_dev_param(void);

void usb_set_adc_vrefcon(void);
void usb_set_rtc(void);

#endif /* SEND_OVER_USB_H_ */
```

SPI_Library.c

```
/*
 * Spi_Library.c
 *
 *  Created on: Oct 20, 2015
 *      Author: cgoss
 *
 *  Modified on: Jan 24, 2018
 *      Author: kdolgikh
 *
 *\file SPI_Library.c
 *\brief Setup and control library for the SPI link to the CC110l
 *
 * Ports and pins for this library are defined in SPI_Library.h, there are ifdef blocks to
target different controllers.
 */

#include <stdint.h>
#include <msp430.h>
#include "CC110l.h"
#include "SPI_Pins.h"
#include "SPI_Library.h"
#include "sleep_timer.h"

// Function to check that address passed into SPI functions is valid
static uint8_t Address_Bad(uint8_t address)
{
	if(address & (BIT7 + BIT6)) // If BIT7 hi throw error
	{
		return 1; // Return 1 if bad
	}
	else
	{
		return 0; // Return 0 if good
	}
}

// Function to spinwait if the radio is not in "ready" state
static void Wait_For_CCWake()
{
	// Spinlock until radio wakes
	while(Port_In & SOMI); // If SOMI is HI, chip is sleeping
}

// It might just be smarter to dispose of the defines and bust this out
// into the init functions, have a specific one for each board. Then we could just
// put the very few remaining outward facing values with the function prototypes
void SPI_Init(void)
{
	// Configure controller IO for SPI pins
	Port_Reg_Sel |= SOMI + SIMO + SPCLK;

	// Configure the GPIO for chip select
	CS_RegisterDir |= CS;

	// CC110l uses active low for chip select, so drive CS HI
	CS_Register |= CS;

	// Set software reset bit to hold USCI module
	USCI_Control_Reg1 = UCSWRST;
```

```
	// Read first edge, clock active HI, MSB bit order, master mode, synchronous (SPI)
	USCI_Control_Reg0 = UCCKPH
                                        + UCMSB
                                        + UCMST
                                        + UCSYNC;

	// Use SMCLK for SPI
	USCI_Control_Reg1 |= UCSSEL_2;

	// MCLK at 20 MHz, run SPI at 1 MHz
	USCI_Modulation_Upper = 0;
	USCI_Modulation_Lower = 20;

	// Clear the status register and interrupt flags before use
	USCI_Status_Reg = 0;
	USCI_Interrupt_Flags = 0;

	// Ready to go, release software reset
	USCI_Control_Reg1 &= ~UCSWRST;
}

uint8_t SPI_Send(uint8_t address, uint8_t value)
{
	uint8_t status; // The return value

	status = USCI_RX_Reg; // Clear the flags

	if(Address_Bad(address))
	{
		return BIT7;
	}

	CS_Register &= ~CS; // Pull CS low

	Wait_For_CCWake(); // Wait for SOMI to go LO

	// TX buffer empties into shift register in one cycle, so we can write twice without
delay
	USCI_TX_Reg = address; // Send address
	while(!(USCI_Interrupt_Flags & USCI_TX_Flag));
	USCI_TX_Reg = value; // Send the actual value,
	// Spinlock until TX finish to ensure CS is held until the transmission is complete
	while(!(USCI_Interrupt_Flags & USCI_RX_Flag));
	status = USCI_RX_Reg; // Read the status byte from the input buffer
	while(!(USCI_Interrupt_Flags & USCI_RX_Flag));


	status = USCI_RX_Reg; // Read the status byte from the input buffer
	CS_Register |= CS; // Pull CS HI

	return status;
}


uint8_t SPI_Read(uint8_t address, uint8_t *out)
{
	uint8_t status;

	*out = USCI_RX_Reg; // Clear the flags
```

```
        // Check for valid address
        if(Address_Bad(address))
        {
                return BIT7;
        }

        CS_Register &= ~CS; // Pull CS low

        Wait_For_CCWake(); // Wait for SOMI to go LO

        USCI_TX_Reg = address | READ_SINGLE; // Send address with MSB HI, indicates read
        while(!(USCI_Interrupt_Flags & USCI_RX_Flag));
        __delay_cycles(10);
        status = USCI_RX_Reg; // Save the status byte
        USCI_TX_Reg = 0xFF; // Gotta send it something to make the SPI hardware run, this will
be ignored by the CC110l

        // Spinlock until RX finish
        while(!(USCI_Interrupt_Flags & USCI_RX_Flag));

        *out = USCI_RX_Reg; // Grab the value

        CS_Register |= CS; // Pull CS HI

        return status;
}

uint8_t SPI_Send_Burst(uint8_t address, uint8_t* value, uint8_t length)
{
        int i;
        uint8_t status;

        if(Address_Bad(address))
        {
                return BIT7;
        }

        CS_Register &= ~CS; // Pull CS low
        Wait_For_CCWake(); // Wait for SOMI to go LO

        USCI_TX_Reg = address | WRITE_BURST; // Send address with burst bit set
        while(!(USCI_Interrupt_Flags & USCI_TX_Flag));
        USCI_TX_Reg = value[0]; // Send first value.

        for(i=1; i< length; i++)
        {
                while(!(USCI_Interrupt_Flags & USCI_TX_Flag)); // Wait for ready
                USCI_TX_Reg = value[i];
        }

        while(!(USCI_Interrupt_Flags & USCI_TX_Flag));
        status = USCI_RX_Reg; // Read the status byte from the input buffer

        while(!(USCI_Interrupt_Flags & USCI_RX_Flag));
        status = USCI_RX_Reg; // Read the status byte from the input buffer

        __delay_cycles(10);
        CS_Register |= CS; // Pull CS HI

        return status;
}
```

```
uint8_t SPI_Read_Burst(uint8_t address, uint8_t* out, uint8_t length)
{
	uint8_t i;
	uint8_t status;

	i = USCI_RX_Reg; // Clear the flags

	if(Address_Bad(address))
	{
		return BIT7;
	}

	CS_Register &= ~CS; // Pull CS low
	Wait_For_CCWake(); // Wait for SOMI to go LO

	USCI_TX_Reg = address | READ_BURST; // Send address with burst bit set and read bit set
	while(!(USCI_Interrupt_Flags & USCI_RX_Flag)); // Wait for ready
	status = USCI_RX_Reg;

	for(i = 0; i < length; i++)
	{
		USCI_TX_Reg = 0xFF;
		while(!(USCI_Interrupt_Flags & USCI_RX_Flag)); // Wait for ready
		out[i] = USCI_RX_Reg;
	}

	CS_Register |= CS; // Pull CS HI
	return status;
}

uint8_t SPI_Strobe(uint8_t strobe, uint8_t FIFO)
{
	uint8_t status;

	status = USCI_RX_Reg; // Clear UCB RX flag

	CS_Register &= ~CS; // Pull CS low
	Wait_For_CCWake(); // Wait for SOMI to go LO

	USCI_TX_Reg = strobe | FIFO; // Pick either the TX or RX FIFO
	while(!(USCI_Interrupt_Flags & USCI_RX_Flag)); // Wait for ready
	status = USCI_RX_Reg;

	CS_Register |= CS; // Pull CS HI
	return status;
}

uint8_t SPI_Read_Status(uint8_t status_reg, uint8_t* out)
{
	uint8_t status;
	status = USCI_RX_Reg;

	CS_Register &= ~CS; // Pull CS low
	Wait_For_CCWake(); // Wait for SOMI to go LO

	USCI_TX_Reg = status_reg | 0xC0; // Send status register address
	while(!(USCI_Interrupt_Flags & USCI_RX_Flag)); // Wait for status byte
	status = USCI_RX_Reg;

	USCI_TX_Reg = 0xFF; // Send garbage
```

```
        while(!(USCI_Interrupt_Flags & USCI_RX_Flag)); // Wait for register value

        CS_Register |= CS; // Pull CS HI

        *out = USCI_RX_Reg;

        return status;
}
```

SPI_Library.h

```
/*
 * Spi_Library.h
 *
 *  Created on: Oct 16, 2015
 *      Author: cgoss
 */

/**
 * \file SPI_Library.h
 * \brief Library for communicating with CC110l over SPI
 *
 */

#include <stdint.h>

#ifndef SPI_LIBRARY_H_
#define SPI_LIBRARY_H_

// Outward facing pins/registers for the radio
/// \name GDO Pins on MSP430
//@{
#define MSP_RX_Pin BIT6                                     ///> Pin that the GDO flagging for
RX is attached to
#define GDO_RX IOCFG0                                   ///> Register controlling the GDO pin used
to signal RX receive.
#define MSP_RX_Port_OUT P1OUT
#define MSP_RX_Port_IFG      P1IFG                  ///> Port interrupt flag
#define MSP_RX_Port_IE       P1IE                   ///> Interrupt enable register
#define MSP_RX_Port_IES P1IES               ///> Interrupt edge select
//@}


/**
 * \brief Initialization function for the SPI link
 *
 * Uses the Port/Pin mapping values in SPI_Pins.h to configure the necessary registers for
 * SPI communication with the CC110l. These mappings are device specific and would need to be
 * changed depending on which MSP430 is being targetted, and which USCI module is being used.
 * The preferred method of handling multiple boards is to use #ifdef blocks targetted at the
 * define value used with the MSP430.h file.
 *
 */

void SPI_Init(void);

/**
 * \brief A function for sending a single byte to the radio
 *
 * Writes a byte to the specified register.
```

```
 *
 * @param address The register in the CC110l that the byte is to be written to.
 * @param value The value to write to the target register.
 * @return The status byte from the CC110l. For a write operation the 4 LSB of the byte will
represent the space left in the TX_FIFO.
 */
uint8_t SPI_Send(uint8_t address, uint8_t value);

/**
 * \brief Reads a single register
 *
 * Fetches the value stored in the specified register. Returns the status byte and copies the
register value into the out parameter
 *
 * @param address The register to be read.
 * @param out Pointer to write the stored register value into.
 * @return The status byte from the CC110l, for a read operation the 4 LSB of the byte
represent the new data in the RX FIFO.
 */
uint8_t SPI_Read(uint8_t address, uint8_t* out);

/**
 * \brief Makes a burst write into a set of registers, beginning with the register pointed by
the address value.
 *
 * The first value in the input array is written into the register at address.
 * The second value is written into the next register, and so on.
 * The status byte is captured from the transmission of the final byte in the burst,
 * and will represent the space left in the TX FIFO just before the final byte is transmitted.
 *
 * @param address The address for the first register to write.
 * @param value An array of values to write into CC110l registers.
 * @param length The number of values that are going to be written to the CC110l.
 * @return The status byte of the CC110l
 */

uint8_t SPI_Send_Burst(uint8_t address, uint8_t* value, uint8_t length);

/**
 * \brief Makes a burst read of the registers in the CC110l.
 *
 * A succession of registers in the CC110l will be copied into the array.
 * The register value at the provided address will be written into the first position
 * of the array, the next register into the second position in the array, and so on.
 *
 * @param address The beginning register address.
 * @param out Pointer to the array the register values are going to be written into.
 * @param length The number of elements to read.
 * @return The CC110l status byte
 */

uint8_t SPI_Read_Burst(uint8_t address, uint8_t* out, uint8_t length);

/**
 * \brief Sends a strobe command to the CC110l
 *
 * The operation of the CC110l is a basic state machine. It will move between
 * states based on either internal events or through strobe commands which are passed to it.
 *
 * @param strobe One of the strobe commands from CC110l.h
 * @param FIFO Get_TX_FIFO or Get_RX_FIFO
```

```
 * @return The CC110l status byte
 */
uint8_t SPI_Strobe(uint8_t strobe, uint8_t FIFO);

/**
 * \brief Reads a value from one of the special status registers in the CC110l
 *
 * The CC110l has a set of special read only status registers. They contain
 * information such as the version number of the chip, the current state machine
 * state, and the FIFO buffer levels.
 *
 * @param status_reg The status register to read the value from.
 * @param out A pointer to the value that the status register should be written into.
 * @return The CC110l status byte.
 */
uint8_t SPI_Read_Status(uint8_t status_reg, uint8_t* out);

#endif /* SPI_LIBRARY_H_ */
```

SPI_Pins.h

```
/*
 * SPI_Pins.h
 *
 *  Created on: Oct 20, 2015
 *      Author: cgoss
 *
 *  Modified on: Jan 24, 2018
 *      Author: kdolgikh
 */

#include "CC110l.h"
#include "driverlib.h"

#ifndef SPI_PINS_H_
#define SPI_PINS_H_

// SPI Pins on MSP430. Port 8
#define SOMI BIT3
#define SIMO BIT2
#define SPCLK BIT1
#define CS BIT0

// SPI Registers
#define Port_Reg_Sel P8SEL                  // Port function select 1
#define Port_Reg_Dir P8DIR                  // Port direction select
#define Port_In P8IN                        // Input values for SPI port
#define CS_Register P8OUT                   // Chip select for the slave device
#define CS_RegisterDir P8DIR                // Direction register for chip select pin
#define USCI_Control_Reg0 UCA1CTL0          // USCI control reg 0
#define USCI_Control_Reg1 UCA1CTL1          // USCI control reg 1
#define USCI_Modulation_Upper UCA1BR1       // Upper byte of modulation value
#define USCI_Modulation_Lower UCA1BR0       // Lower byte of modulation vale
#define USCI_Status_Reg UCA1STAT            // Status register
#define USCI_Interrupt_Flags UCA1IFG        // Interrupt flag register
#define USCI_TX_Reg UCA1TXBUF               // Transmit buffer
#define USCI_RX_Reg UCA1RXBUF               // Receive buffer

// Register masks
#define USCI_RX_Flag UCRXIFG                // Mask for RX complete
```

```
#define USCI_TX_Flag UCTXIFG                // Mask for TX complete

// GDO Ports on MSP430
#define MSP_RX_Port_DIR             P1DIR   // Direction register
#define MSP_RX_Port_OUT             P1OUT   // Out value register
#define MSP_RX_Port_REN             P1REN   // Resistor enable register
#define MSP_RX_Port_IN              P1IN    // In value register


#endif /* SPI_PINS_H_ */
```

usbEventHandling.c

```
/* --COPYRIGHT--,BSD
 * Copyright (c) 2016, Texas Instruments Incorporated
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * *  Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * *  Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * *  Neither the name of Texas Instruments Incorporated nor the names of
 *    its contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
 * OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 * --/COPYRIGHT--*/
/*
 * ======== usbEventHandling.c ========
 * Event-handling placeholder functions.
 * All functios are called in interrupt context.
 */
#include "driverlib.h"

#include "USB_API/USB_Common/device.h"
#include "USB_API/USB_Common/defMSP430USB.h"
#include "USB_config/descriptors.h"
#include "USB_API/USB_Common/usb.h"

#include "send_over_usb.h"
#include "adc_driver.h"
#include "states.h"
#include "measure.h"
```

```
#ifdef _CDC_
#include "USB_API/USB_CDC_API/UsbCdc.h"
#endif

#ifdef _HID_
#include "USB_API/USB_HID_API/UsbHid.h"
#endif

#ifdef _MSC_
#include "USB_API/USB_MSC_API/UsbMsc.h"
#endif

#ifdef _PHDC_
#include "USB_API/USB_PHDC_API/UsbPHDC.h"
#endif

extern device_state dev_state;
uint8_t dev_state_old; // retain device state prior USB connection

/*
 * If this function gets executed, it's a sign that the output of the USB PLL has failed.
 * returns TRUE to keep CPU awake
 */
uint8_t USB_handleClockEvent ()
{
    //TO DO: You can place your code here
    return (TRUE); //return TRUE to wake the main loop (in the case the CPU slept before
interrupt)
}

/*
 * If this function gets executed, it indicates that a valid voltage has just been applied to
the VBUS pin.
 * returns TRUE to keep CPU awake
 */
uint8_t USB_handleVbusOnEvent ()
{
       if (RTCPS1CTL&RT1PSIE)
              RTCPS1CTL &= ~RT1PSIE; // Disable RT1PS interrupt, because no need in meas rate
when connected

       USBconnected = TRUE;

       if (!adcWorkInProgress) // Check if adc is working
       {
              dev_state = usb_connected;
              return (TRUE); //return TRUE to wake the main loop (in the case the CPU slept
before interrupt)
       }
       else
              return (FALSE); // If adc is working, don't wake up here
}

/*
 * If this function gets executed, it indicates that a valid voltage has just been removed
from the VBUS pin.
 * returns TRUE to keep CPU awake
 */
uint8_t USB_handleVbusOffEvent ()
{
    //TO DO: You can place your code here
```

```
        UCS_turnOffXT2();        // disable XT2 crystal
        USBconnected = FALSE;
        firstMeasurement = TRUE; // Will initiate measuring after the first connection to USB
(for setup)

        if(!KeepData)
                mem_counters_cleanup(); // Will start writing from the beginning of memory
        else
                index=TS_W_CRC_NUM_BYTES; // Will loose everything that is in RAM (up to 10
measurements)

        dev_state = adc_measuring; // After disconnect start with adc_cal state
        P4OUT &= ~BIT6;            // switch power to battery
        if(!adcWorkInProgress)
                return (TRUE);     //return TRUE to wake the main loop (in the case the CPU
slept before interrupt)
        else
                return (FALSE);
}

/*
 * If this function gets executed, it indicates that the USB host has issued a USB reset event
to the device.
 * returns TRUE to keep CPU awake
 */
uint8_t USB_handleResetEvent ()
{
    //TO DO: You can place your code here
        dev_state = usb_reset;
    return (TRUE); //return TRUE to wake the main loop (in the case the CPU slept before
interrupt)
}

/*
 * If this function gets executed, it indicates that the USB host has chosen to suspend this
device after a period of active
 * operation.
 * returns TRUE to keep CPU awake
 */
uint8_t USB_handleSuspendEvent ()
{
    //TO DO: You can place your code here
        P4OUT &= ~BIT6;              // switch power to battery
        dev_state = usb_suspended;
        return (TRUE);               //return TRUE to wake the main loop (in the case the CPU
slept before interrupt)
}

/*
 * If this function gets executed, it indicates that the USB host has chosen to resume this
device after a period of suspended
 * operation.
 * returns TRUE to keep CPU awake
 */
uint8_t USB_handleResumeEvent ()
{
    //TO DO: You can place your code here
        P4OUT |= BIT6;               // switch power to USB
        dev_state = usb_enumerated;
        return (TRUE);               //return TRUE to wake the main loop (in the case the CPU
slept before interrupt)
```

```
}

/*
 * If this function gets executed, it indicates that the USB host has enumerated this device :
 * after host assigned the address to the device.
 * returns TRUE to keep CPU awake
 */
uint8_t USB_handleEnumerationCompleteEvent ()
{
    //TO DO: You can place your code here
       P4OUT |= BIT6;                       // switch power to USB
       dev_state = usb_enumerated;
    return (TRUE);                          //return TRUE to wake the main loop (in the case
the CPU slept before interrupt)
}

/* This #define can be used to reduce latency times during crystal/PLL
 * startups; see the Programmer's Guide for more information.
 */
#ifdef USE_TIMER_FOR_RESUME

/*
 * Indicates a USB resume is in progress, XT2 has been turned on, and the USB
 * API requires the application to call USB_enable_PLL() after XT2 has
 * stabilized.
 */
void USB_handleCrystalStartedEvent(void)
{
    // Unlike other events, these don't have a return value to wake main()
}

/*
 * Indicates a USB resume is in progress, the PLL has been turned on, and the
 * USB API requires the application to call USB_enable_final() after the PLL
 * has locked.
 */
void USB_handlePLLStartedEvent(void)
{
    // Unlike other events, these don't have a return value to wake main()
}
#endif

#ifdef _CDC_

/*
 * This event indicates that data has been received for interface intfNum, but no data receive
operation is underway.
 * returns TRUE to keep CPU awake
 */
uint8_t USBCDC_handleDataReceived (uint8_t intfNum)
{
    //TO DO: You can place your code here
       USB_DataReceived_event = TRUE;
    return (TRUE);                              //return FALSE to go asleep after interrupt
(in the case the CPU slept before
                                                //interrupt)
}

/*
 * This event indicates that a send operation on interface intfNum has just been completed.
 * returns TRUE to keep CPU awake
```

```
 */
uint8_t USBCDC_handleSendCompleted (uint8_t intfNum)
{
    //TO DO: You can place your code here

    return (TRUE); //return FALSE to go asleep after interrupt (in the case the CPU slept
before //interrupt)
}

/*
 * This event indicates that a receive operation on interface intfNum has just been completed.
 */
uint8_t USBCDC_handleReceiveCompleted (uint8_t intfNum)
{
    //TO DO: You can place your code here

       return (FALSE); //return FALSE to go asleep after interrupt (in the case the CPU slept
before //interrupt)
}

/*
 * This event indicates that new line coding params have been received from the host
 */
uint8_t USBCDC_handleSetLineCoding (uint8_t intfNum, uint32_t lBaudrate)
{
    //TO DO: You can place your code here

    return (FALSE); //return FALSE to go asleep after interrupt (in the case the CPU slept
before //interrupt)
}

/*
 * This event indicates that new line state has been received from the host
 */
uint8_t USBCDC_handleSetControlLineState (uint8_t intfNum, uint8_t lineState)
{
       return FALSE;
}

#endif //_CDC_

/*-------------------------------------------------------------------------+
 | End of source file                                                      |
 +-------------------------------------------------------------------------*/
/*------------------------ Nothing Below This Line ------------------------*/
//Released_Version_5_20_06_02
```

sleep_timer.c

```
/*
 * sleep_timer.c
 *
 *  Created on: Apr 9, 2017
 *      Author: kdolgikh
 */

#include "driverlib.h"
#include <stdint.h>
#include "sleep_timer.h"
#include "states.h"
```

```
volatile uint8_t timer_event = FALSE;
uint8_t delay_enable = FALSE;


void Sleep_Timer_Fast(uint32_t cycles)
{
	// 1 cycle = 1/4096 or 0.244 ms or 4882 MCLK cycles.
	// If MCU wake-up time is slow (165 us), having 1 cycle of smaller than 165 us is not
reasonable, use _delay_cycles(x);
	// account for extra 165 us required to wake-up, if necessary.
	// If MCU wake-up time is fast (3 us), TA2 source frequency can be increased to 32768
(1 cycle is 30.5 us)
	// or even changed to SMCLK if smaller than 30.5 us sleep duration is required.

	uint16_t i;
	uint32_t rollovers;
	uint16_t Sleep_Cycles;

	rollovers = (cycles >> 16);		// Upper word is number of rollovers
	Sleep_Cycles = (cycles & 0xFFFF);	// Lower word is number of cycles to run

	if (Sleep_Cycles == 0) // If 0 (if rollovers >0), then TAIFG and CCRIFG will occur
simultaneously,
		Sleep_Cycles = 1; // and, because CCRIFG has higher priority, the rollover count
will not be set // before evaluating the for loop. As the result the device will sleep for 1
rollover longer. // When sleep_cycles = 1, CCRIFG will occur later than TAIFG. Also,
protection from 0 input.

	TA2CCR1 = Sleep_Cycles; // Set the CCR to wait the number of cycles given

	TA2CTL |= MC__CONTINUOUS; //Start timer

	LPM3;

	for (i = 0; i < rollovers && timer_event; i++)
	{
		timer_event = FALSE;
		LPM3;
	}

	TA2CTL ^= MC__CONTINUOUS; // Stop timer
	TA2R = 0; // Clear TA1R
}
```

sleep_timer.h

```
/*
 * sleep_timer.h
 *
 *  Created on: Apr 9, 2017
 *      Author: kdolgikh
 */

#ifndef SLEEP_TIMER_H_
#define SLEEP_TIMER_H_

#include <stdint.h>

#define TRUE   1
```

```
#define FALSE  0

extern volatile uint16_t Fast_Timer_Rollover_Count;
extern volatile uint8_t timer_event;
extern volatile uint8_t rtc_event;
extern uint8_t delay_enable;

/**
 * \brief      Puts MCU to sleep for the specified number of ACLK cycles.
 *
 * @param      cycles  The number of cycles to sleep for, in terms of ACLK ticks
 */
void Sleep_Timer_Fast(uint32_t cycles);

#endif /* SLEEP_TIMER_H_ */
```

states.h

```
/*
 * states.h
 *
 *  Created on: Dec 5, 2016
 *      Author: kdolgikh
 */

/**
 * \file states.h
 * \brief Contains information on different states of the device and its modules
 */

#ifndef STATES_H_
#define STATES_H_

#define STATUS_INTERRUPTED   2

#define WRITE_SUCCESS 0x10
#define WRITE_FAIL    0x11

#define CAL_REQD      1
#define CAL_NOT_REQD  0

typedef enum {
       adc_read_single_reg,  ///< Read single register
       adc_write_single_reg, ///< Write single register
       adc_read_all_reg,     ///< Read all registers
       adc_rx_conv_data      ///< Receive conversion data
} adc_comm_state;


typedef enum {
       usb_enumerated,///< The device is connected to PC and ready to TX/RX data over USB
       usb_not_enum,  ///< For USB states other than enumerated and suspended
       usb_connected, ///< USB cable has been connected to the device
       adc_set,       ///< Set parameters of the ADC
       adc_cal,       ///< Calibration is in progress
       adc_measuring, ///< Measure is in progress
       adc_rst,       ///< Reset ADC
       dev_mem_full, ///< Device memory is full
       dev_malfunction,///< Device broken
       parameter_error,///< Error in operational parameter
```

```
        usb_reset,     ///< Add code to handle USB reset
        usb_suspended  ///< The device has been suspended by the host
} device_state;


typedef enum {
        not_exit,
        exit1,
        exit2,
        exit3,
        exit4,
        exit5,
        exit6,
        exit7,
        exit8
} exit_point;

typedef enum {
        rtc_operation,
        bulk_operation
} operation_mode;

#endif /* STATES_H_ */
```

ucs_functions.c

```
/*
 * ucs_functions.c
 *
 *  Created on: Nov 30, 2016
 *      Author: kdolgikh
 */

/** \file ucs_functions.c
 *      \brief Contains a set of UCS functions
 */

#include "msp430f5659.h"
#include "ucs_functions.h"

void enable_XT1_RTC (void) // RTC and XT1 should be set up before clearing LOCKBACK, see p130
UG
{
        RTCPS1CTL |= RT1IP__256; // Set RTC prescaler to generate 0.5 Hz intervals
        clear_XT1_OFFG();        // Allow XT1 to settle
        UCSCTL6 &= ~XT1DRIVE_3;  // Set the lowest drive setting for 32.768 Hz crystal
}

void clear_XT1_OFFG (void)
{
    do
    {
        UCSCTL7 &= ~(XT1LFOFFG + DCOFFG); // Clear XT1 and DCO fault flags
        SFRIFG1 &= ~OFIFG; // Clear oscillator fault flag, see page 82 UG
    }
    while (SFRIFG1&OFIFG); // Loop until OFIFG is cleared
}

uint8_t clear_XT1_OFFG_timeout (uint32_t timeout)
{
```

```
    do {
        UCSCTL7 &= ~(XT1LFOFFG + DCOFFG); // Clear XT1 and DCO fault flags
        SFRIFG1 &= ~OFIFG; // Clear oscillator fault flag, see page 82 UG
    } while ((SFRIFG1&OFIFG) && --timeout);  // Do until timeout has expired or flag has been
cleared

    if (timeout)
        return (UCS_STATUS_OK); // Timeout has not expired
    else
        return (UCS_STATUS_ERROR); // Timeout has expired
}

void disable_XT1 (void)
{
    UCSCTL6 |= XT1OFF; // Disable XT1
}

void clear_lockback_batbackup (void)
{
       while (BAKCTL&LOCKBAK)
       {
              BAKCTL &= ~LOCKBAK; // Clear LOCKBACK flag to release XT1 for operation
       }
}

void enable_XT2 (void)
{
       P7SEL |= BIT2; // Choose port 7.2 for XT2 operation, page 100 DS, page 164 UG
       UCSCTL6 &= ~XT2OFF; // Enable XT2CLK
       clear_XT2_OFFG(); // Allow XT2 to settle
       UCSCTL6 &= ~XT2DRIVE_3; // Set the lowest drive setting for 4 MHz crystal
}

void clear_XT2_OFFG (void)
{
    do
    {
       UCSCTL7 &= ~XT2OFFG; // Clear XT2 fault flag
       SFRIFG1 &= ~OFIFG; // Clear oscillator fault flag, see page 82 UG
    }
    while (SFRIFG1&OFIFG); // Loop until OFIFG is cleared
}

uint8_t clear_XT2_OFFG_timeout (uint32_t timeout)
{
    do {
        UCSCTL7 &= ~XT2OFFG; // Clear XT2 fault flag
        SFRIFG1 &= ~OFIFG;   // Clear oscillator fault flag, see page 82 UG
    } while ((SFRIFG1&OFIFG) && --timeout); // Do until timeout has expired or flag has been
cleared

    if (timeout)
        return (UCS_STATUS_OK); // Timeout has not expired
    else
        return (UCS_STATUS_ERROR); // Timeout has expired
}

void disable_XT2 (void)
{
    UCSCTL6 |= XT2OFF; // Disable XT2 oscillator
}
```

ucs_functions.h

```
/*
 * ucs_functions.h
 *
 *  Created on: Nov 30, 2016
 *      Author: kdolgikh
 */

/** \file ucs_functions.h
 *     \brief Contains a set of UCS functions
 */

#ifndef UCS_FUNCTIONS_H_
#define UCS_FUNCTIONS_H_

#include <stdint.h>

#define UCS_STATUS_OK     0
#define UCS_STATUS_ERROR  1

/** \brief Enable XT1 crystal and RTC
 *
 *     This function is used on device startup.
 *     DCO fault flag will also appear at this time.
 *
 *     Execution flow:
 *     1. XT1 is enabled by default after reset. Wait until XT1 settles.
 *     2. Set XT1 drive to the smallest allowed value.
 */
void enable_XT1_RTC (void);

/** \brief Clear XT1 related oscillator fault flags
 *
 *     This function is used whenever it's necessary to clear XT1 faults
 */
void clear_XT1_OFFG (void);

/** \brief Clear XT1 related oscillator fault flags with a specified timeout
 *
 *     This function is used whenever it's necessary to clear XT1 faults with the ability
 *     to exit the function upon timeout expiration
 *
 *     @param timeout       Amount of time to try clearing oscillator fault flag
 *     @return              Return status of the clear operation: success or fail
 */
uint8_t clear_XT1_OFFG_timeout (uint32_t timeout);


/** \brief Disable XT1
 *
 *     This function is used whenever it's necessary to disable XT1
 */
void disable_XT1 (void);

/** \brief Clear LOCKBACK flag of the battery backup system
 *
 *     Until cleared, battery backup system doesn't allow XT1 to operate
```

```
 */
void clear_lockback_batbackup (void);


/** \brief Enable XT2 crystal
 *
 *     This function is used when USB VBUS appears
 *
 *     Execution flow:
 *     1. Enable XT2 and wait until it settles.
 *     2. Set XT2 drive to the smallest allowed value.
 */
void enable_XT2 (void);

/** \brief Clear XT2 related oscillator fault flags
 *
 *     This function is used whenever it's necessary to clear XT2 faults
 */
void clear_XT2_OFFG (void);

/** \brief Clear XT2 related oscillator fault flags with a specified timeout
 *
 *     This function is used whenever it's necessary to clear XT1 faults with the ability
 *     to exit the function upon timeout expiration
 *
 *     @param timeout       Amount of time to try clearing oscillator fault flag
 *     @return              Return status of the clear operation: success or fail
 */
uint8_t clear_XT2_OFFG_timeout (uint32_t timeout);

/** \brief Disable XT2
 *
 *     This function is used whenever it's necessary to disable XT2
 */
void disable_XT2 (void);

#endif /* UCS_FUNCTIONS_H_ */
```

Appendix G

The User Interface Code

cli.m

```
global is_nice;
is_nice=0;

global ch_num;
ch_num=0;

global year;
global month;
global day;
global hour;
global minute;
global second;
global set_time_fail_count;
global rx_completed;
global set_time_in_progress;
year = 0;
month = 0;
day = 0;
hour = 0;
minute=0;
second=0;
set_time_fail_count=0;
rx_completed=0;
set_time_in_progress=0;

global num_segm_bulk;
num_segm_bulk = uint8(30);

global data_rate;
data_rate = 2000;

global meas_rate;
meas_rate = uint32(10);

global op_mode;
op_mode = uint8(0); % 0 is rtc mode, 1 is bulk mode

% USB commands
global set_meas_rate;
global get_conv_result;
global get_data_rate;
global set_data_rate;
global get_num_bulk_segm;
global set_num_bulk_segm;
global get_vrefcon;
global set_vrefcon;
global get_ofc;
global get_fsc;
global get_dev_state;
global set_dev_state;
global get_rtc;
global set_rtc;
global set_sysocal_en;
global get_sysocal_en;
```

```
global set_delay_en;
global get_delay_en;
global set_meas_start;
global get_meas_rate;
global set_keep_data;
global get_keep_data;
global get_cal_en;
global set_cal_en;
global get_op_mode;
global set_op_mode;

get_conv_result = uint8(127);          % 0x7F
get_data_rate = uint8(3);              % 0x03
set_data_rate = uint8(5);              % 0x05
get_num_bulk_segm = uint8(7);          % 0x07
set_num_bulk_segm = uint8(9);          % 0x09
get_vrefcon = uint8(10);               % 0x0A
set_vrefcon = uint8(11);               % 0x0B
get_ofc = uint8(12);                   % 0x0C
get_fsc = uint8(13);                   % 0x0D
get_dev_state = uint8(14);             % 0x0E
set_dev_state = uint8(15);             % 0x0F
get_rtc = uint8(17);                   % 0x11
set_rtc = uint8(18);                   % 0x12
get_sysocal_en = uint8(19);            % 0x13
set_sysocal_en = uint8(20);            % 0x14
get_delay_en = uint8(21);              % 0x15
set_delay_en = uint8(22);              % 0x16
set_meas_rate = uint8(23);             % 0x17
set_meas_start = uint8(24);            % 0x18
get_meas_rate = uint8(25);             % 0x19
set_keep_data = uint8(26);             % 0x1A
get_keep_data = uint8(27);             % 0x1B
set_cal_en = uint8(28);                % 0x1C
get_cal_en = uint8(29);                % 0x1D
get_op_mode = uint8(30);               % 0x1E
set_op_mode = uint8(31);               % 0x1F


break_event = 1;
return_key='';
accepted_val = 1;

com_ports = seriallist; % provides the list of connected COM ports

while(accepted_val)
    disp('#Available COM ports: ');
    disp(com_ports);
    choose_dev_prompt = '#Choose COM port by entering its name (e.g. COM1)\n#';
    choose_dev = input(choose_dev_prompt,'s');
    for i=1:numel(com_ports)
        if strcmp(choose_dev,com_ports(i))
            accepted_val = 0;
            s = serial(com_ports(i));
        end
    end

    if accepted_val == 1
        disp('#Entered COM port is invalid, please retry');
        pause(1)
    end
end
get_adc_param(s,'get data rate');
```

```
% start CLI
prompt = '#';

while(break_event)
    pause(1)
    user_input = input(prompt,'s');
    switch (user_input)
        case 'get conv result'
            request_num_bytes(s);
        case 'get data rate'
            get_adc_param(s,user_input);
        case 'get meas rate'
            get_measurement_rate(s);
        case 'get num bulk'
            get_adc_param(s,user_input);
        case 'get adc vrefcon'
            get_adc_param(s,user_input);
        case 'get dev state'
            get_adc_param(s,user_input);
        case 'get sysocal en'
            get_adc_param(s,user_input);
        case 'get delay en'
            get_adc_param(s,user_input);
        case 'get keep data'
            get_adc_param(s,user_input);
        case 'get cal en'
            get_adc_param(s,user_input);
        case 'get op mode'
            get_adc_param(s,user_input);
        case 'get ofc reg'
            get_adc_cal_register(s,user_input);
        case 'get fsc reg'
            get_adc_cal_register(s,user_input);
        case 'get rtc'
            get_rt_clock(s);
        case 'set meas rate'
            set_measurement_rate(s);
        case 'set meas start'
            set_measure_start(s);
        case 'set data rate'
            set_adc_data_rate(s);
        case 'set num bulk'
            set_num_of_bulk_segments(s);
        case 'set adc vrefcon'
            set_adc_vrefcon_reg(s);
        case 'set dev state'
            set_device_state(s);
        case 'set sysocal en'
            set_sysocal_enable(s);
        case 'set delay en'
            set_delay_enable(s);
        case 'set rtc'
             set_rt_clock(s);
        case 'set time'
            set_time(s);
        case 'set keep data'
            set_keep_data_in_memory(s);
        case 'set cal en'
            set_cal_enable(s);
        case 'set op mode'
            set_operational_mode(s);
        case 'help'
```

```
            cli_help();
        case 'help vrefcon'
            cli_help_vrefcon();
        case 'help dev state'
            cli_help_dstate();
        case 'abort'
            fclose(s);
            disp('#Serial port is closed');
        case 'exit'
            break_event = 0;
        case return_key
            disp('#');
        otherwise
            disp('#Unrecognized command');
    end
end

fclose(s);
delete(s);
clear
clc
```

request_num_bytes.m

```
function request_num_bytes(obj)
% This function requests amount of data to be received from the Logger

    global get_conv_result;

    % Length of the buffer for num_bytes ( 1 byte ACK + 4 bytes num_bytes)
    num_bytes_buff_lgth = 5;

    % input buffer size for receiving num conv bytes from the device
    % Buffer size is larger than BytesAvailableFcnCount by 1
    obj.InputBufferSize =  num_bytes_buff_lgth + 1;

    % specify number of bytes to be received
    % 5 bytes, ACK and one uint32_t should be received
    obj.BytesAvailableFcnCount = num_bytes_buff_lgth;

    % interrupt when receive the specified number of bytes
    obj.BytesAvailableFcnMode = 'byte';

    % specify function to be executed when received BytesAvailableFcnCount
    obj.BytesAvailableFcn = {@num_bytes_RXed_callback};

    fopen(obj);       % open serial port
    disp('#Serial port is opened');
    readasync(obj);   % callbacks (interrupts) work only in async mode

    % request number of conversions
    fwrite(obj,get_conv_result);
    disp('#Get conv result is sent');
end
```

num_bytes_RXed_callback.m

```
function num_bytes_RXed_callback(obj,~,~)
    % Function receives num of conv result bytes, prepares data buffer
    % to receive data and sends ACK to the device to start TXing conv data

    global get_conv_result;

    ack_bytes = 1;  % number of ack bytes that will be sent to the host

    param = fread(obj,obj.BytesAvailable,'uint8');

    fclose(obj);
    disp('#Serial port is closed');

    % the first byte is ACK
    if param(1) == get_conv_result
        % use bytes 2 to 5 to obtain num bytes
        % use typecast to convert four uint8 to one uint32
        % data is received MSB first
        param = swapbytes(typecast(uint8((param(2:5))),'uint32'));
        disp(['#Num conv bytes:   ',num2str(param)]);

        % input buffer size for receiving num conv bytes from the device
        % buffer size is larger than BytesAvailableFcnCount by 1
        obj.InputBufferSize =  param + ack_bytes + 1;

        % specify number of bytes to be received
        % + 1 accounts for get_conv_result comand sent back as ACK from the
        % device
        obj.BytesAvailableFcnCount = param + ack_bytes;

        % interrupt when receive the specified number of bytes
        obj.BytesAvailableFcnMode = 'byte';

        % specify function to be executed when received BytesAvailableFcnCount
        obj.BytesAvailableFcn = {@data_received_callback,param};

        % if required number of bytes is not received within 1 min,
        % go to get_buffer_callback
        obj.TimerPeriod = 60; % seconds
        obj.TimerFcn = @get_buffer_callback;

        fopen(obj);       % open serial port
        disp('#Serial port is opened');
        readasync(obj);   % callbacks (interrupts) work only in async mode

        % request conv data
        fwrite(obj,get_conv_result);
        disp('#Get conv result is sent');
    else
        disp('#Invalid ACK is received. Please, retry');
    end
end
```

data_received_callback.m

```
function data_received_callback (obj,~,num_bytes)
    % This callback function is executed when finished receiving data from
    % the device.

    global get_conv_result;
    global meas_rate;
    global op_mode;
    global num_segm_bulk;

    % read conversion result (cr)
    cr = fread(obj,obj.BytesAvailable,'uint8');

    fclose(obj);
    disp('#Serial port is closed');

    if cr(1) == get_conv_result
        disp('#Conversion result is successfully received');
        cr=cr(2:end);   % remove ack
        if ~op_mode % rtc mode
            ProcessInputData(cr, num_bytes, meas_rate);
        else        % bulk mode
            ProcessInputBulkData(cr,num_bytes, num_segm_bulk);
        end
    else
        disp('#Invalid ACK is received. Please, retry');
    end
end
```

ProcessInputData.m

```
function ProcessInputData(cr, num_bytes, meas_rate)
    % Function processes and presents conversion result to the user

    global data_rate;
    global op_mode;

    segment_size = 512;
    num_meas_per_sect = 10;
    millenium = 2000;
    num_channels = 16;

    % need to explicitly convert to appropriate data types, otherwise won't work
    num_bytes=double(num_bytes);
    cr=uint8(cr);

    % Data organization
    % Byte 1: ACK
    % Byte 2 - 512, 513 - 1024 etc - data segments.
    % Each data segment has:
        % Byte 1 - 5:   timestamp, byte 1 - year, bytes 2-5 - MDHMS
        % Byte 6:       result of CRC check for time stamp
        % Byte 7:       0x00
        % Byte 8 - 55:  conversion result for 16 channels
        % Byte 56:      result of CRC check for conv results
        % Byte 57:      0x00

    %% Generate time stamp vector

    % number of sectors in received data
    % round for the case the data was requested before segment was filled
```

```
num_sectors = fix(num_bytes/segment_size);
not_full_sector = rem(num_bytes,segment_size);

% position of the time stamp CRC check result minus 1
% to account for initial indexing starting at 1
ts_crc_position = 5;
bad_data = uint8(189); % 0xBD
%correct_data = uint8(205); % 0xCD

% number of time stamps in received data
num_ts = num_sectors;

% number of measurements in complete segments
num_meas = num_ts* num_meas_per_sect;

if (not_full_sector > 0)
    num_ts = num_ts + 1;
    num_meas_not_full_sect = fix(not_full_sector/50);
    num_meas = num_meas + num_meas_not_full_sect; % total num of meas
end

% create array for time stamps
year = uint32(zeros(1,num_ts));
mon = uint32(zeros(1,num_ts));
day = uint32(zeros(1,num_ts));
hour = uint32(zeros(1,num_ts));
minute = uint32(zeros(1,num_ts));
sec = uint32(zeros(1,num_ts));

segment_ts_err = zeros(1,num_ts);   % order of the segment with ts CRC error

% generate time stamps from input data
i=1; j=1;
while i <= num_ts
    if cr(j+ts_crc_position) ~= bad_data
        year(i) = uint32(cr(j)) + millenium;
        time_uint32 = swapbytes(typecast(cr((j+1):(j+4)),'uint32'));
        [mon(i),day(i),hour(i),minute(i),sec(i)] = ...
                        extract_time(time_uint32);
        % generate datetime values
        t(i) = datetime(year(i),mon(i),day(i),hour(i),minute(i),sec(i));
    else
        segment_ts_err(i) = fix(j/segment_size) + 1;
        t(i) = NaT;
    end
    i = i + 1;
    j = j + segment_size;
end

% Calculate number of TS CRC errors for CRC num display
crc_tcount=0;
for i=1:length(t)
   if isnat(t(i))
       crc_tcount=crc_tcount+1;
   end
end

% Generate full time stamp vector
[D,H,M,S]=transform_meas_rate(meas_rate);

% In case all time stamps are invalid, use fake time stamp. This is
% ! temporary for testing only !
```

```
if crc_tcount == length(t)
   temp_t=datetime(clock);
   t(1) = temp_t - num_meas * (days(D)+hours(H)+minutes(M)+seconds(S));
   t_failed = 'True';
else
   t_failed = 'False';
end

err_free_ts_ind = find(segment_ts_err==0);  % index of elements without error
err_free_ts_ind = min(err_free_ts_ind);     % choose the minimal value of index
                                            % (the first err free index)

if num_ts > 1
    if err_free_ts_ind > 1 % error free index not the first one
        %do backward counting from index to the first element,
        i=(err_free_ts_ind-1);
        while i>=1
            t(i)=t(i+1)-num_meas_per_sect*(days(D)+...
                hours(H)+minutes(M)+seconds(S));
            i=i-1;
        end
    end

    ts(1)=t(1);
    for i=2:1:num_meas
        ts(i)= ts(i-1) + days(D)+hours(H)+minutes(M)+seconds(S);
    end
else
    ts(1)=t;
    for i=2:1:num_meas
        ts(i)= ts(i-1) + days(D)+hours(H)+minutes(M)+seconds(S);
    end
end

%% Process data in spare bytes (5 bytes in the end of each segment)

%% Generate data matrix

% convert individual bytes into conversion results of 24 bits for all 16 ch
% can't use swapbytes(typecast(,'uint32')) easily because I need 4 bytes,
% and data is 3 bytes. Will use own function "transform_conv_result."

% position of the conv result MSB for 16 channels starting at ch 1
conv_res_pos = [8,11,14,17,20,23,26,29,...
            32,35,38,41,44,47,50,53];

data_spread = 50;   % spread of crc as well as conv result data values
data_crc_pos = 56;  % position of the first data CRC byte
spare_bytes = 5;

meas_result=double(zeros(num_channels,num_meas));

% convert individual bytes into conversion results of 24 bits for all 16 ch
% and replace results having CRC error with NaN
```

```
    for i=1:num_channels
    k=conv_res_pos(i);
    m=data_crc_pos;
    count=0;
    for  j=1:num_meas
        meas_result(i,j) = transform_conv_result(cr(k),cr(k+1),cr(k+2));
        if (cr(m)==bad_data)
            meas_result(i,j)=NaN;
        end
        count=count+1;
        if (count == 10)
            count = 0;
            k = k +data_spread +spare_bytes +conv_res_pos(1) - 1;
            m = m +data_crc_pos +spare_bytes +1;
        else
            k = k + data_spread;
            m = m + data_spread;
        end
    end
    end

    % Display number of CRC errors
    % Number of TS CRC errors was determined during TS processing stage
    % Determine num of data CRC errors
    temp_var = meas_result(1,1:end);
    crc_dcount=0;
    for i=1:length(temp_var)
       if isnan(temp_var(i))
           crc_dcount=crc_dcount+1;
       end
    end

    disp(['#Number of time stamps (TS):     ',num2str(num_ts)]);
    disp(['#Number of TS CRC errors:        ',num2str(crc_tcount)]);
    disp(['#Unable to reconstruct TS:       ',t_failed]);
    disp(['#Number of measurements:         ',num2str(num_meas)]);
    disp(['#Number of data CRC errors:      ',num2str(crc_dcount)]);

    %% Process and present conv result

    [temperature,voltage,Rth] = PresentConvResult(num_meas,...
                                num_channels,meas_result,ts,op_mode);

    %% save workspace and files
    DataRate = num2str(data_rate);
    TimeStamp = datestr(datetime(clock),'yymmddTHHMM');
    FileName = strcat(TimeStamp,'_All_Ch_DR',DataRate);
    save(strcat(FileName,'_code.txt'),'meas_result','-ascii','-double','-tabs');
    save(strcat(FileName,'_volt.txt'),'voltage','-ascii','-double','-tabs');
    save(strcat(FileName,'_rth.txt'),'Rth','-ascii','-double','-tabs');
    save(strcat(FileName,'_temp.txt'),'temperature','-ascii','-double','-tabs');
    save(strcat(FileName,'_ts.mat'),'ts','-mat');
    save(strcat(FileName,'.mat'));
end
```

transform_meas_rate.m

```
function [Day,Hour,Min,Sec] = transform_meas_rate(meas_rate)

% Transforms measurement rate in seconds into day-hour-min-sec format
    seconds_min = 60;
    seconds_hour = 3600;
    seconds_day = 86400;

    Day = fix(meas_rate/seconds_day); % number of full days
    part_day = rem(meas_rate,seconds_day);

    Hour = fix(part_day/seconds_hour); % value greater than 0 indicates partial day
    part_hour = rem(part_day,seconds_hour);

    Min = fix(part_hour/seconds_min);
    Sec = rem(part_hour,seconds_min);
end
```

transform_conv_result.m

```
function [conv_result] = transform_conv_result(byte1,byte2,byte3)
% Transforms conversion result of 24 bytes into one double
% Double format is chosen because it can hold NaN

conv_result = double(byte1)*2^16 + double(byte2)*2^8 + double(byte3);
end
```

PresentConvResult.m

```
function [temperature,voltage,Rth] =
PresentConvResult(num_meas,num_channels,meas_result,ts,op_mode)

    global is_nice;
    global fig_number;
    fig_number = 1;

    ch_num_array = 1:num_channels;

    prompt = '#Type the name of the thermistor calibration file\n#';
    user_input = input(prompt,'s');
    therm_rt_data = load(user_input);

    % "Nice" board will have additional calibration factor(mux Ron)
    accepted_value = 1;
    while(accepted_value)
        prompt = '#Is "Nice"? y/n\n#';
        nice = input(prompt,'s');
        if strcmp(nice,'y')
            is_nice = 1;
            accepted_value = 0;
        else
            if strcmp(nice,'n')
                is_nice = 0;
                accepted_value = 0;
            else
            disp('#Error. Type either "y" or "n"');
            end
        end
    end
```

```
    accepted_value = 1;
    while(accepted_value)
        prompt = '#Skip saving individual channels data? y/n\n#';
        skip = input(prompt,'s');
        if strcmp(skip,'y')
            skip_saving = 1;
            accepted_value = 0;
        else
            if strcmp(skip,'n')
                skip_saving = 0;
                accepted_value = 0;
            else
            disp('#Error. Type either "y" or "n"');
            end
        end
    end

    temperature=zeros(num_channels,num_meas);
    voltage=zeros(num_channels,num_meas);
    Rth=zeros(num_channels,num_meas);
    exit=0;
    while(~exit)
        prompt = '#Enter "1 to 16" to choose channel, "all" for all channels or "stop"
to return\n#';
        user_input = input(prompt,'s');
        if strcmp(user_input,'all')
            prompt = '#Use plot(p) or scatter(s)?\n#';
            sop = input(prompt,'s');
            if strcmp(sop,'p')
                is_plot = 1;
            else
                is_plot = 0;
            end

            for i=1:num_channels
                [temperature(i,1:end), voltage(i,1:end), Rth(i,1:end)] = ...

ProcessConvResult(meas_result(i,1:end),i,therm_rt_data,ts,1,skip_saving,op_mode);
            end

            legend_val=[];
            for i=1:length(ch_num_array)
                legend_val_next = strcat("Ch ",num2str(ch_num_array(i)));
                legend_val = [legend_val, legend_val_next];
            end

            prompt = '#Plot measured voltage y/n?\n#';
            ui_v = input(prompt,'s');
            if strcmp(ui_v,'y')
                figure(fig_number)
                if is_plot
                    plot(ts,voltage);
                else
                    for i=1:num_channels
                        hold on
                        scatter(ts,voltage(i,1:end));
                        hold off
                    end
                end
                title 'Measured thermistor voltage for all channels'
                xlabel 'Time'; ylabel 'Thermistor voltage, V';
                legend(legend_val,'Location','eastoutside');
```

```
        axis tight
        fig_number = fig_number + 1;

        prompt = '#Show statistics for voltage data y/n?\n#';
        ui_s = input(prompt,'s');
        if strcmp(ui_s,'y')
            analyse_logger_data_all_ch(ch_num_array,voltage,0);
        end
    end

    prompt = '#Plot thermistor resistance y/n?\n#';
    ui_r = input(prompt,'s');
    if strcmp(ui_r,'y')
        figure(fig_number)
        if is_plot
            plot(ts,Rth);
        else
            for i=1:num_channels
                hold on
                scatter(ts,Rth(i,1:end));
                hold off
            end
        end
        title 'Measured thermistor resistance for all channels'
        xlabel 'Time'; ylabel 'Thermistor resistance, Ohm';
        legend(legend_val,'Location','eastoutside');
        axis tight
        fig_number = fig_number + 1;

        prompt = '#Show statistics for resistance data y/n?\n#';
        ui_s = input(prompt,'s');
        if strcmp(ui_s,'y')
            analyse_logger_data_all_ch(ch_num_array,Rth,0);
        end
    end

    prompt = '#Plot thermistor temperature y/n?\n#';
    ui_t = input(prompt,'s');
    if strcmp(ui_t,'y')
        figure(fig_number)
        if is_plot
            plot(ts,temperature);
        else
            for i=1:num_channels
                hold on
                scatter(ts,temperature(i,1:end));
                hold off
            end
        end
        title 'Measured temperature for all channels'
        xlabel 'Time'; ylabel 'Temperature, C';
        legend(legend_val,'Location','eastoutside');
        axis tight
        fig_number = fig_number + 1;

        prompt = '#Show statistics for temperature data y/n?\n#';
        ui_s = input(prompt,'s');
        if strcmp(ui_s,'y')
            analyse_logger_data_all_ch(ch_num_array,temperature,0);
        end
    end

else
```

```
            if find(ch_num_array==str2double(user_input))
                ch_num = str2double(user_input);

ProcessConvResult(meas_result(ch_num,1:end),ch_num,therm_rt_data,ts,0,skip_saving,op_m
ode);
            else
                if strcmp(user_input,'stop')
                    exit = 1;
                else
                    disp('#Entered channel number is invalid. Valid values are "1 -
16"');
                end
            end
        end
    end
end
```

ProcessConvResult.m

```
function [temperature, v, Rth] =
ProcessConvResult(conv_result,channel,therm_rt_data,time_stamp,skip_display,skip_savin
g,op_mode)
% Processes conversion result for a single channel
% Returns measured temperature, voltage, resistance

    %% Defines
    global fig_number;
    global is_nice;
    global data_rate;

    %% Get measured voltage
    vref=2.0464;                        % reference voltage
    v=conv_result*vref/(2^23-1);        % measured voltage for the case FS = 0 to +Vref

    %% Get measured thermistor resistance

    RR=116662.3;                % reference resistance, Ohm
    Rmux=4;                     % mux Ron, see plot on page 7 ADG706 UG
    Rmeas=RR*conv_result./(2^23 - 1 - conv_result); % measured resistance

    if is_nice
        Rth=Rmeas - Rmux;
    else
        Rth=Rmeas;
    end

    %% Get measured temperature

    % thermistor calibration curves
    y = therm_rt_data(1:end,2); % resistance
    x = therm_rt_data(1:end,1); % temperature
    temperature=interp1(y,x,Rth,'pchip'); % interpolated temperature values

    %% Plot
    if (~skip_display)
        prompt = '#Use plot(p) or scatter(s)?\n#';
        sop = input(prompt,'s');
        if strcmp(sop,'p')
            is_plot = 1;
        else
            is_plot = 0;
        end
```

```
prompt = '#Plot measured voltage y/n?\n#';
ui_v = input(prompt,'s');
if strcmp(ui_v,'y')
    figure(fig_number)
    if is_plot
        fig = plot(time_stamp,v);
    else
        fig = scatter(time_stamp,v);
    end
    title(strcat('Measured thermistor voltage for ch',num2str(channel)));
    xlabel 'Time'; ylabel 'Input voltage, V';
    axis tight
    saveas(fig,strcat('Device_Ch',num2str(channel),'_volt.png'));
    fig_number = fig_number + 1;

    prompt = '#Show statistics for voltage data y/n?\n#';
    ui_s = input(prompt,'s');
    if strcmp(ui_s,'y')
        analyse_logger_data(v,skip_display);
    end

    if op_mode % show histogram only in bulk_mode
        prompt = '#Plot histogram y/n?\n#';
        ui_s = input(prompt,'s');
        if strcmp(ui_s,'y')
            figure(fig_number)
            fig = histogram(v);
            str = {strcat('Ch',num2str(channel)),...
                strcat('MeasRate=',num2str((1000/data_rate)),'ms'),...
                strcat(num2str(length(v)),' Samples')};
            dim = [.15 .6 .3 .3];
            annotation('textbox',dim,'String',str,'FitBoxToText','on');
            title 'Measured Voltage Distribution';
            xlabel 'Voltage, V'; ylabel 'Counts';
            saveas(fig,strcat('Device_Ch',num2str(channel),'_volt_hist.png'));
            fig_number = fig_number + 1;
        end
    end
end

prompt = '#Plot thermistor resistance y/n?\n#';
ui_r = input(prompt,'s');
if strcmp(ui_r,'y')
    figure(fig_number)
    if is_plot
        fig = plot(time_stamp,Rth);
    else
        fig = scatter(time_stamp,Rth);
    end
    title(strcat('Measured thermistor resistance for ch',num2str(channel)));
    xlabel 'Time'; ylabel 'Resistance, Ohm';
    axis tight
    saveas(fig,strcat('Device_Ch',num2str(channel),'_res.png'));
    fig_number = fig_number + 1;

    prompt = '#Show statistics for resistance data y/n?\n#';
    ui_s = input(prompt,'s');
    if strcmp(ui_s,'y')
        analyse_logger_data(Rth,skip_display);
    end
```

```
        if op_mode % show histogram only in bulk_mode
            prompt = '#Plot histogram y/n?\n#';
            ui_s = input(prompt,'s');
            if strcmp(ui_s,'y')
                figure(fig_number)
                fig = histogram(Rth/1e3);
                str = {strcat('Ch',num2str(channel)),...
                    strcat('MeasRate=',num2str((1000/data_rate)),'ms'),...
                    strcat(num2str(length(Rth)),' Samples')};
                dim = [.15 .6 .3 .3];
                annotation('textbox',dim,'String',str,'FitBoxToText','on');
                title 'Measured Resistance Distribution';
                xlabel 'Resistance, KOhm'; ylabel 'Counts';
                saveas(fig,strcat('Device_Ch',num2str(channel),'_res_hist.png'));
                fig_number = fig_number + 1;
            end
        end
    end

    prompt = '#Plot thermistor temperature y/n?\n#';
    ui_t = input(prompt,'s');
    if strcmp(ui_t,'y')
        figure(fig_number)
        if is_plot
            fig = plot(time_stamp,temperature);
        else
            fig = scatter(time_stamp,temperature);
        end
        title(strcat('Measured temperature for ch',num2str(channel)));
        xlabel 'Time'; ylabel 'Temperature, C';
        axis tight
        saveas(fig,strcat('Device_Ch',num2str(channel),'_temp.png'));
        fig_number = fig_number + 1;

        prompt = '#Show statistics for temperature data y/n?\n#';
        ui_s = input(prompt,'s');
        if strcmp(ui_s,'y')
            analyse_logger_data(temperature,skip_display);
        end

        if op_mode % show histogram only in bulk_mode
            prompt = '#Plot histogram y/n?\n#';
            ui_s = input(prompt,'s');
            if strcmp(ui_s,'y')
                figure(fig_number)
                fig = histogram(temperature);
                str = {strcat('Ch',num2str(channel)),...
                    strcat('MeasRate=',num2str((1000/data_rate)),'ms'),...
                    strcat(num2str(length(temperature)),' Samples')};
                dim = [.15 .6 .3 .3];
                annotation('textbox',dim,'String',str,'FitBoxToText','on');
                title 'Measured Temperature Distribution';
                xlabel 'Temperature, C'; ylabel 'Counts';
                saveas(fig,strcat('Device_Ch',num2str(channel),'_temp_hist.png'));
                fig_number = fig_number + 1;
            end
        end
    end
end

%% Save data
% generate file name
if (~skip_saving)
```

```
        ChanNum = num2str(channel);
        DataRate = num2str(data_rate);
        TimeStamp = datestr(datetime(clock),'yymmddTHHMM');

        if is_nice
            FileName = strcat(TimeStamp,'_N_Ch',ChanNum,'_DR',DataRate);
            FileNameTemp =
strcat(TimeStamp,'_N_Ch',ChanNum,'_DR',DataRate,'_temp.txt');
            FileNameCode =
strcat(TimeStamp,'_N_Ch',ChanNum,'_DR',DataRate,'_code.txt');
            FileNameVolt =
strcat(TimeStamp,'_N_Ch',ChanNum,'_DR',DataRate,'_volt.txt');
        else
            FileName = strcat(TimeStamp,'_U_Ch',ChanNum,'_DR',DataRate);
            FileNameTemp =
strcat(TimeStamp,'_U_Ch',ChanNum,'_DR',DataRate,'_temp.txt');
            FileNameCode =
strcat(TimeStamp,'_U_Ch',ChanNum,'_DR',DataRate,'_temp.txt');
            FileNameVolt =
strcat(TimeStamp,'_N_Ch',ChanNum,'_DR',DataRate,'_volt.txt');
        end

        % Save the result to file
        save(FileNameTemp,'temperature','-ascii','-double','-tabs');
        save(FileNameCode,'conv_result','-ascii','-double','-tabs');
        save(FileNameVolt,'v','-ascii','-double','-tabs');
        save(strcat(FileName,'.mat'));
    end
end
```

analyse_logger_data.m

```
function analyse_logger_data(data,skip_disp)
    data_avg = mean(data,'omitnan');
    data_max = max(data,[],'omitnan');
    data_min = min(data,[],'omitnan');
    data_range = data_max - data_min;
    data_std = std(data,'omitnan');
    if (~skip_disp)
        disp(['Average data:       ',num2str(data_avg)]);
        disp(['Max data:           ',num2str(data_max)]);
        disp(['Min data:           ',num2str(data_min)]);
        disp(['Data range:         ',num2str(data_range)]);
        disp(['Data std deviation: ',num2str(data_std)]);
    end
end
```

analyse_logger_data_all_ch.m

```
function [metrics]=analyse_logger_data_all_ch(not_outliers,data,skip_display)
% Function analyzes data from all channels of the logger

   metrics = struct('max_max',0,...     % max value for all ch
                    'min_min',0,...     % min value for all ch
                    'range_max',0,...   % range for all ch
                    'max_avg',0,...     % max among avg values for individual channels
                    'min_avg',0,...     % min among avg values for individual channels
                    'max_std',0,...     % max among std values for all channels
                    'min_std',0,...     % min among std values for all channels
                    'all_ch_avg',length(data));
```

```
    data_avg = zeros(1,length(not_outliers));
    data_max = zeros(1,length(not_outliers));
    data_min = zeros(1,length(not_outliers));
    data_range = zeros(1,length(not_outliers));
    data_std = zeros(1,length(not_outliers));

    for i=not_outliers
        data_avg(i) = mean(data(i,1:end),'omitnan');
        data_max(i) = max(data(i,1:end),[],'omitnan');
        data_min(i) = min(data(i,1:end),[],'omitnan');
        data_range(i) = data_max(i) - data_min(i);
        data_std(i) = std(data(i,1:end),'omitnan');
    end

    metrics.max_max = max(data_max(not_outliers));    % max data for all ch
    metrics.min_min = min(data_min(not_outliers));    % min data for all ch
    metrics.range_max = metrics.max_max - metrics.min_min;
    metrics.max_avg = max(data_avg(not_outliers));    % max avg for all ch
    metrics.min_avg = min(data_avg(not_outliers));    % min avg for all ch
    metrics.max_std = max(data_std(not_outliers));    % max std for all ch
    metrics.min_std = min(data_std(not_outliers));    % min std for all ch

    % average value of all channels for each individual measurement
    for i = 1:length(data(1,1:end))
        metrics.all_ch_avg(i) = mean(data(not_outliers(1:end),i));
    end

    if (~skip_display)
        disp(['Max value for all channels:            ',num2str(metrics.max_max)]);
        disp(['Min value for all channels:            ',num2str(metrics.min_min)]);
        disp(['Range for all channels:                ',num2str(metrics.range_max)]);
        disp(['Max avg value for all channels:        ',num2str(metrics.max_avg)]);
        disp(['Min avg value for all channels:        ',num2str(metrics.min_avg)]);
        disp(['Max std deviation for all channels:    ',num2str(metrics.max_std)]);
        disp(['Min std deviation for all channels:    ',num2str(metrics.min_std)]);
    end
end
```

ProcessInputBulkData.m

```
function ProcessInputBulkData(cr,num_bytes,num_segm_bulk)

    global data_rate;
    global op_mode;
    segment_size = 512;
    meas_per_segment = 170;
    crc_bytes = 2;

    %% process input data

    % !Important! It's crucial that the program receives the complete
    % blocks of data of size num_segm_bulk*num_segm for each channel.
    % There might be less channels than 16, but the blocks must be
    % complete, or the code won't work. It is a future task to create the
    % code that will handle incomplete blocks (which is needed for the case
    % when USB is inserted at any time)

   % determine num channels
    num_channels = num_bytes/(num_segm_bulk*segment_size);
    bad_data = 189; % 0xBD
    conv_result_crc = zeros(1,num_segm_bulk*num_channels);
    j=1;
    for i=1:num_segm_bulk*num_channels
```

```
        conv_result_crc(i) = cr((i*segment_size - 1));
        if conv_result_crc(i) == bad_data
            cr(j:(j + segment_size - 3)) = NaN;
        end
        j= j + segment_size;
    end

    % find if bad_data is present and display it
    crc_bd = find(conv_result_crc == bad_data);
    disp(['#Number of CRC errors: ',num2str(length(crc_bd))]);
    if ~isempty(crc_bd)
        disp('#Invalid data was replaced with NaN');
    end

    a=[];
    for i=1:num_segm_bulk*num_channels
        if (i==1)
            a_next = cr(i:(i*segment_size - crc_bytes));
        else
            a_next = cr(((i-1)*segment_size+1):(i*segment_size - crc_bytes));
        end
        a=[a; a_next];
    end

    code_bulk = zeros(1,(length(a)/3));
    k=1;    % index of a
    i=1;    % index of code_bulk
    while (k <= (length(a)-2))
        code_bulk(i) = a(k)*2^16 + a(k+1)*2^8 + a(k+2); % shifting operation equals
multiplication
        k=k+3;
        i=i+1;
    end

    %% separate different channels data
    code = [];
    i=1;
    for j=1:num_channels
        code_next = code_bulk(i:j*num_segm_bulk*meas_per_segment);
        code = [code; code_next];
        i = i + num_segm_bulk*meas_per_segment;
    end

    tstep = 1/data_rate;     % sampling time
    time=0:tstep:(tstep*(length(code)-1));

    %% process and present the result
   [temperature,voltage,Rth] =
PresentConvResult(num_segm_bulk*meas_per_segment,num_channels,code,time,op_mode);

    %% save data
    DataRate = num2str(data_rate);
    TimeStamp = datestr(datetime(clock),'yymmddTHHMM');
    FileName = strcat(TimeStamp,'_All_Ch_DR',DataRate);
    save(strcat(FileName,'_code.txt'),'code','-ascii','-double','-tabs');
    save(strcat(FileName,'_volt.txt'),'voltage','-ascii','-double','-tabs');
    save(strcat(FileName,'_rth.txt'),'Rth','-ascii','-double','-tabs');
    save(strcat(FileName,'_temp.txt'),'temperature','-ascii','-double','-tabs');
    save(strcat(FileName,'_time.mat'),'time','-mat');
    save(strcat(FileName,'.mat'));
end
```

get_bugger_calback.m

```
function get_buffer_callback(obj,~)
    disp('Conv result was not received within 1 minute');
    num_bytes = num2str(obj.BytesAvailable);
    disp(['#Number of received bytes: ',num_bytes]);
    data = fread(obj,obj.BytesAvailable,'uint8');
    save(strcat('data_',num_bytes,'_bytes.mat'),'data');
    disp('#Received data is saved to file')
    fclose(obj);
    disp('#Serial port is closed');
end
```

get_adc_param.m

```
function get_adc_param(obj, usr_input)

    global get_data_rate;
    global get_num_bulk_segm;
    global get_vrefcon;
    global get_dev_state;
    global get_sysocal_en;
    global get_delay_en;
    global get_keep_data;
    global get_cal_en;
    global get_op_mode;

    if (strcmp(usr_input,'get data rate'))
        command = get_data_rate;
    else
        if (strcmp(usr_input,'get num bulk'))
        command = get_num_bulk_segm;
        else
            if (strcmp(usr_input,'get adc vrefcon'))
            command = get_vrefcon;
            else
                if (strcmp(usr_input,'get dev state'))
                   command = get_dev_state;
                else
                    if (strcmp(usr_input,'get sysocal en'))
                        command = get_sysocal_en;
                    else
                        if (strcmp(usr_input,'get delay en'))
                            command = get_delay_en;
                        else
                            if (strcmp(usr_input,'get keep data'))
                                command = get_keep_data;
                            else
                                if (strcmp(usr_input,'get cal en'))
                                    command = get_cal_en;
                                else
                                    if (strcmp(usr_input,'get op mode'))
                                        command = get_op_mode;
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    end
```

```
    result_str = strcat('#',usr_input,' command is sent');

    obj.InputBufferSize = 3;

    % specify number of bytes to be received
    obj.BytesAvailableFcnCount = 2; % 2 bytes should be RXed: ACK and param

    % interrupt when receive the specified number of bytes
    obj.BytesAvailableFcnMode = 'byte';

    % specify function to be executed when received BytesAvailableFcnCount
    obj.BytesAvailableFcn = {@param_received_callback,command};

    fopen(obj);       % open serial port
    disp('#Serial port is opened');
    readasync(obj);   % callbacks (interrupts) work only in async mode

    fwrite(obj,command);
    disp(result_str);
end
```

param_received_callback.m

```
function param_received_callback(obj,~,command)

    global num_segm_bulk;
    global data_rate;
    global op_mode;

    global get_data_rate;
    global get_num_bulk_segm;
    global get_vrefcon;
    global get_dev_state;
    global get_sysocal_en;
    global get_delay_en;
    global get_keep_data;
    global get_cal_en;
    global get_op_mode;

    % Data rates
    dr5 = 0;
    dr10 = 1;
    dr20 = 2;
    dr40 = 3;
    dr80 = 4;
    dr160 = 5;
    dr320 = 6;
    dr640 = 7;
    dr1000 = 8;
    dr2000 = 9;

    % vrefcon values
    vrefcon_vref_off = 0;
    vrefcon_vref_on = 32;
    vrefcon_vref_alt = 64;

    % device states
    get_dev_state_disabled = 0;
    adc_set = 3;
    adc_cal = 4;
    adc_measure = 5;
    adc_rst = 6;
```

```
    dev_memory_full = 7;
    dev_malfunction = 8;
    param_error = 9;

    %read received param
    param = fread(obj,obj.BytesAvailable,'uint8');

    fclose(obj);
    disp('#Serial port is closed');

    if param(1) == command
            if command == get_data_rate
                switch(param(2))
                    case dr5
                       param_disp = 5;
                    case dr10
                        param_disp = 10;
                    case dr20
                        param_disp = 20;
                    case dr40
                        param_disp = 40;
                    case dr80
                        param_disp = 80;
                    case dr160
                        param_disp = 160;
                    case dr320
                        param_disp = 320;
                    case dr640
                        param_disp = 640;
                    case dr1000
                        param_disp = 1000;
                    case dr2000
                        param_disp = 2000;
                end
                data_rate = param_disp;
            else
                if command == get_num_bulk_segm
                    param_disp = num2str(param(2));
                    num_segm_bulk = param(2);
                else
                    if command == get_vrefcon
                        switch(param(2))
                            case vrefcon_vref_off
                                param_disp = 'vref_off';
                            case vrefcon_vref_on
                                param_disp = 'vref_on';
                            case vrefcon_vref_alt
                                param_disp = 'vref_alt';
                        end
                    else
                        if command == get_dev_state
                            switch(param(2))
                                case get_dev_state_disabled
                                    param_disp = 'get device state function is
disabled';
                                case adc_set
                                    param_disp = 'adc_set';
                                case adc_cal
                                    param_disp = 'adc_cal';
                                case adc_measure
                                    param_disp = 'adc_measure';
                                case adc_rst
                                    param_disp = 'adc_rst';
```

```
                                case dev_malfunction
                                    param_disp = 'dev_malfunction';
                                case dev_memory_full
                                    param_disp = 'dev_mem_full';
                                case param_error
                                    param_disp = 'dev_param_error';
                            end
                        else
                            if command == get_sysocal_en || command == get_delay_en
||...command == get_keep_data || command == get_cal_en
                                switch(param(2))
                                    case 0
                                        param_disp = 'False';
                                    case 1
                                       param_disp = 'True';
                                end
                            else
                                if command == get_op_mode
                                    op_mode = param(2);
                                    switch(param(2))
                                        case 0
                                            param_disp = 'rtc_mode';
                                        case 1
                                            param_disp = 'bulk_mode';
                                    end
                                end
                            end
                        end
                    end
                end

                switch (command)
                    case get_data_rate
                        command_str ='data_rate';
                    case get_num_bulk_segm
                        command_str ='num_bulk';
                    case get_vrefcon
                        command_str ='adc_vrefcon';
                    case get_dev_state
                        command_str ='dev_state';
                    case get_sysocal_en
                        command_str ='sysocal_en';
                    case get_delay_en
                        command_str ='delay_en';
                    case get_keep_data
                        command_str = 'keep_data';
                    case get_cal_en
                        command_str = 'cal_en';
                    case get_op_mode
                        command_str = 'get_op_mode';
                end

                if command == get_data_rate
                    disp(['#Requested ',command_str,' value is   ',num2str(param_disp)]);
                 else
                    disp(['#Requested ',command_str,' value is   ',param_disp]);
                end
            else
                disp('#Invalid ACK is received. Please, retry');
            end
end
```

get_measurement_rate.m

```
function get_measurement_rate(obj)

    global get_meas_rate;

    obj.InputBufferSize = 6;

    % 5 bytes (1 byte ACK plus 4 bytes meas rate) should be received
    obj.BytesAvailableFcnCount = 5;

    % interrupt when receive the specified number of bytes
    obj.BytesAvailableFcnMode = 'byte';

    % specify function to be executed when received BytesAvailableFcnCount
    obj.BytesAvailableFcn = {@meas_rate_received_callback,get_meas_rate};

    fopen(obj);       % open serial port
    disp('#Serial port is opened');
    readasync(obj);   % callbacks (interrupts) work only in async mode

    % request number of conversions
    fwrite(obj,get_meas_rate);
    disp('#Get meas rate command is sent');
end
```

meas_rate_received_callback.m

```
function meas_rate_received_callback(obj,~,command)

    data = fread(obj,obj.BytesAvailable,'uint8');
    fclose(obj);
    disp('#Serial port is closed');

    global meas_rate;

    if data(1) == command
        meas_rate = swapbytes(typecast(uint8(data(2:end)),'uint32'));
        [D,H,M,S] = transform_meas_rate(meas_rate);
        disp('#The device measurement rate is:');
        disp([num2str(D),' days ',num2str(H),' hours ',num2str(M),...
            ' minutes ',num2str(S),' seconds ']);
    else
        disp('#Invalid ACK is received. Please, retry')
    end
end
```

get_adc_cal_register.m

```
function get_adc_cal_register(obj, user_input)

    global get_ofc;
    global get_fsc;

    if (strcmp(user_input,'get ofc reg'))
        command = get_ofc;
    else if (strcmp(user_input,'get fsc reg'))
        command = get_fsc;
        end
    end
```

```
    obj.InputBufferSize = 5;

    % 4 bytes (1 byte ACK plus 3 bytes register) should be received
    obj.BytesAvailableFcnCount = 4;

    % interrupt when receive the specified number of bytes
    obj.BytesAvailableFcnMode = 'byte';

    % specify function to be executed when received BytesAvailableFcnCount
    obj.BytesAvailableFcn = {@cal_reg_received_callback,command};

    fopen(obj);       % open serial port
    disp('#Serial port is opened');
    readasync(obj);   % callbacks (interrupts) work only in async mode

    % request number of conversions
    fwrite(obj,command);
    disp('#Command is sent');
end
```

cal_reg_received_callback.m

```
function cal_reg_received_callback(obj,~,command)

    calreg = fread(obj,obj.BytesAvailable,'uint8');
    fclose(obj);
    disp('#Serial port is closed');

    if calreg(1) == command
        % shifting operation equals multiplication
        cal_register = calreg(2)*2^16 + calreg(3)*2^8 + calreg(4);
        disp(['#Requested calibration register value is ',num2str(cal_register)]);
    else
        disp('#Invalid ACK is received. Please, retry');
    end
end
```

get_rt_clock.m

```
function get_rt_clock(obj)

    global get_rtc;
    obj.InputBufferSize = 8;

    % 7 bytes (1 byte ACK plus 6 bytes clock) should be received
    obj.BytesAvailableFcnCount = 7;

    % interrupt when receive the specified number of bytes
    obj.BytesAvailableFcnMode = 'byte';

    % specify function to be executed when received BytesAvailableFcnCount
    obj.BytesAvailableFcn = {@clock_received_callback,get_rtc};

    fopen(obj);       % open serial port
    disp('#Serial port is opened');
    readasync(obj);   % callbacks (interrupts) work only in async mode

    % request number of conversions
    fwrite(obj,get_rtc);
```

```
    disp('#Get RT clock command is sent');
end
```

clock_received_callback.m

```
function clock_received_callback(obj,~,command)
    clock = fread(obj,obj.BytesAvailable);
    fclose(obj);
    disp('#Serial port is closed');

    global year;
    global month;
    global day;
    global hour;
    global minute;
    global second;
    global rx_completed;
    global set_time_in_progress;

    if clock(1) == command
        year=clock(2)+2000;
        month=clock(3);
        day=clock(4);
        hour=clock(5);
        minute=clock(6);
        second=clock(7);
       if ~set_time_in_progress
           t=datetime(year,month,day,hour,minute,second);
           disp('#The device clock is:');
           disp(t);
       end
    else
        disp('#Invalid ACK is received. Please, retry')
    end

    rx_completed=1;
end
```

set_measurement_rate.m

```
function set_measurement_rate (obj)

    global set_meas_rate;
    global data_rate;
    global meas_rate;

    empty_array='';

    disp('#Enter measurement rate in days, hours, months and seconds');
    disp('#Number of seconds should always be divisible by 2');

    accepted_value = 1;
    while(accepted_value)
        prompt = '#Enter number of days: ';
        Days = input(prompt,'s');
        if(find(isstrprop(Days,'digit')==0))
            disp('#Invalid value. Days should be a number');
        else
            if strcmp(Days,empty_array)
                disp('#Invalid value. Days should be a number');
            else
```

```
            days = str2double(Days);
            if days < 0 || days > 49710 % num of days in 2^32 seconds
                disp('#Invalid value. Days should lie within 0 - 49710 range');
            else
                accepted_value=0;
            end
        end
    end
end

accepted_value = 1;
while(accepted_value)
    prompt = '#Enter number of hours: ';
    Hours = input(prompt,'s');
    if(find(isstrprop(Hours,'digit')==0))
        disp('#Invalid value. Hours should be a number');
    else
        if strcmp(Hours,empty_array)
            disp('#Invalid value. Hours should be a number');
        else
            hours = str2double(Hours);
            if hours <0 || hours >23
                disp('#Invalid value. Hours should lie within 0 - 23 range');
            else
                accepted_value=0;
            end
        end
    end
end

accepted_value = 1;
while(accepted_value)
    prompt = '#Enter number of minutes: ';
    Minutes = input(prompt,'s');
    if(find(isstrprop(Minutes,'digit')==0))
        disp('#Invalid value. Minutes should be a number');
    else
        if strcmp(Minutes,empty_array)
            disp('#Invalid value. Minutes should be a number');
        else
            minutes = str2double(Minutes);
            if minutes <0 || minutes >59
                disp('#Invalid value. Minutes should lie within 0 - 59 range');
            else
                accepted_value=0;
            end
        end
    end
end

accepted_value = 1;
while(accepted_value)
    prompt = '#Enter number of seconds: ';
    Seconds = input(prompt,'s');
    if(find(isstrprop(Seconds,'digit')==0))
        disp('#Invalid value. Seconds should be a number');
    else
        if strcmp(Seconds,empty_array)
            disp('#Invalid value. Seconds should be a number');
        else
            seconds = str2double(Seconds);
            if seconds <0 || seconds >59
                disp('#Invalid value. Seconds should lie within 0 - 59 range');
```

```
                else
                    if rem(seconds,2) ~= 0
                        disp('#Invalid value. Seconds should be divisible by 2')
                    else
                        if data_rate == 5
                                if seconds < 12 && days == 0 && hours == 0 && minutes
== 0
                                    disp('#Invalid value. For data rate 5 sps min
seconds is 12');
                                else
                                    accepted_value=0;
                                end
                            else
                                if data_rate == 10
                                    if seconds < 6 && days == 0 && hours == 0 && minutes
== 0
                                        disp('#Invalid value. For data rate 10 sps min
seconds is 6');
                                    else
                                        accepted_value=0;
                                    end
                                else
                                    if data_rate == 20 || data_rate == 40
                                        if seconds < 4 && days == 0 && hours == 0 &&
minutes == 0
                                            disp('#Invalid value. For data rates 20 and 40
sps min seconds is 4');
                                        else
                                            accepted_value=0;
                                        end
                                    else
                                        if data_rate == 80 || data_rate == 160 ||
data_rate == 320 ||...
                                           data_rate == 640 || data_rate == 1000 ||
data_rate == 2000
                                           if seconds < 2 && days == 0 && hours == 0 &&
minutes == 0
                                                disp('#Invalid value. For data rates 80 -
2000 sps min seconds is 2');
                                           else
                                                accepted_value=0;
                                           end
                                        end
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    end

    meas_rate =
swapbytes(uint32(transform_meas_rate_sec(days,hours,minutes,seconds)));

    mr = typecast(meas_rate,'uint8');

    meas_rate = swapbytes(meas_rate);

    message =  cat(2,set_meas_rate,mr);

    send_set_cmd(obj,set_meas_rate,message);
end
```

send_set_cmd.m

```
function send_set_cmd(obj, command, message)

% Function sends set commands over serial port.
% Message contains both the command code and data
% to be sent. Command code itself is used in ack_callback
% and in displayed result.

    global set_meas_rate;
    global set_data_rate;
    global set_num_bulk_segm;
    global set_vrefcon;
    global set_dev_state;
    global set_rtc;
    global set_sysocal_en;
    global set_delay_en;
    global set_meas_start;
    global set_keep_data;
    global set_cal_en;
    global set_op_mode;

    switch (command)
        case set_meas_rate
            command_str ='set meas rate';
        case set_meas_start
            command_str ='set meas start';
        case set_data_rate
            command_str ='set data rate';
        case set_num_bulk_segm
            command_str ='set num bulk';
        case set_vrefcon
            command_str ='set_adc_vrefcon';
        case set_dev_state
            command_str ='set dev state';
        case set_sysocal_en
            command_str ='set sysocal en';
        case set_delay_en
            command_str ='set delay en';
        case set_rtc
            command_str ='set rtc';
        case set_keep_data
            command_str ='set keep data';
        case set_cal_en
            command_str ='set cal en';
        case set_op_mode
            command_str ='set op mode';
    end

    result_str = strcat('#',command_str,' command is sent');

    obj.InputBufferSize = 2;

    % specify number of bytes to be received
    obj.BytesAvailableFcnCount = 1; % only 1 byte ACK should be received

    % interrupt when receive the specified number of bytes
    obj.BytesAvailableFcnMode = 'byte';

    % specify function to be executed when received BytesAvailableFcnCount
    obj.BytesAvailableFcn = {@ack_received_callback,command};

    fopen(obj);       % open serial port
```

```
    disp('#Serial port is opened');

    readasync(obj);   % callbacks (interrupts) work only in async mode

    % send value of the data rate
    fwrite(obj,message);
    disp(result_str);
end
```

ack_received_callback.m

```
function ack_received_callback(obj,~,command)

    ack = fread(obj,obj.BytesAvailable,'uint8');
    fclose(obj);
    disp('#Serial port is closed');

    if ack == command
        disp('#Parameter is set successfully')
    else
        disp('#Invalid ACK is received. Please, retry')
    end

    global rx_completed;
    rx_completed = 1;
end
```

set_measure_start.m

```
function set_measure_start(obj)

    global set_meas_start;

    % options:
    % start immediately
    % start at 00 ( xx min 00 sec, xx hour 00 min 00 sec, etc): will implement later
    % start at specific time: will implement later
    % start with delay -> enter delay: will implement later

    % Start with delay is implemented in set_delay_en,
    % although the delay itself cannot be set.
    % Will remove set_delay_en and replace with this function later.

    start_zero = 0;
    disp('#Choose measurement start options:');
    accepted_value = 1;
    while(accepted_value)
    prompt = '#Start immediately? Enter y/n ';
    start_imm = input(prompt,'s');
        if strcmp(start_imm,'y')
            accepted_value=0;
            start_imm = 1;
        else
            if strcmp(start_imm,'n')
               accepted_value=0;
               start_imm = 0;
            else
                disp('#Invalid value. Enter y for "yes" or n for "no"');
             end
        end
```

```
    end

    if ~start_imm
        accepted_value = 1;
        while(accepted_value)
        prompt = '#Start at zero clock? Enter y/n ';
        start_zero = input(prompt,'s');
            if strcmp(start_zero,'y')
                accepted_value=0;
                start_zero = 1;
            else
                if strcmp(start_zero,'n')
                    accepted_value=0;
                    start_zero = 0;
                else
                    disp('#Invalid value. Enter y for "yes" or n for "no"');
                end
            end
        end
    end

    message = [set_meas_start,uint8(start_imm),uint8(start_zero)];

    send_set_cmd(obj,set_meas_start,message);
end
```

set_adc_data_rate.m

```
function set_adc_data_rate(obj)

    global data_rate;
    global set_data_rate;

    % Data rates
    dr5 = 0;
    dr10 = 1;
    dr20 = 2;
    dr40 = 3;
    dr80 = 4;
    dr160 = 5;
    dr320 = 6;
    dr640 = 7;
    dr1000 = 8;
    dr2000 = 9;

    accepted_value = 1;

    while(accepted_value)
        disp('#Valid datarates: 5, 10, 20, 40, 80, 160, 320, 640, 1000, 2000');
        prompt = '#Enter data rate: ';
        dr = input(prompt);
          if dr==5 || dr==10 || dr==20 || dr==40|| dr==80 || dr==160 || dr==320 ||
dr==640 || dr==1000 || dr==2000
            accepted_value = 0;
          else
            disp('#Entered data rate is invalid');
            pause(1)
          end
    end

    data_rate = dr;
```

```
    switch(dr)
      case(5)
        dr = dr5;
      case(10)
        dr = dr10;
      case(20)
        dr = dr20;
      case(40)
        dr = dr40;
      case(80)
        dr = dr80;
      case(160)
        dr = dr160;
      case(320)
        dr = dr320;
      case(640)
        dr = dr640;
      case(1000)
        dr = dr1000;
      case(2000)
        dr = dr2000;
    end

    message =  [set_data_rate, dr];

    send_set_cmd(obj,set_data_rate,message);
end
```

set_num_of_bulk_segments.m

```
function set_num_of_bulk_segments(obj)

    global set_num_bulk_segm;
    global num_segm_bulk;

    accepted_value = 1;

    while(accepted_value)
        disp('#Valid number of bulk segments is 1 - 48');
        prompt = '#Enter number of conversions: ';
        number_conv = input(prompt);
        if ((number_conv > 0) && (number_conv <  49))
            accepted_value = 0;
        else
            disp('#Error. Number of segments must be within 1 - 48 range');
            pause(1)
        end
    end

    num_segm_bulk = number_conv;

    message =  [set_num_bulk_segm, number_conv];

    send_set_cmd(obj,set_num_bulk_segm,message);
end
```

set_adc_vrefcon_reg.m

```
function set_adc_vrefcon_reg(obj)

    global set_vrefcon;

    % vrefcon values
    vrefcon_vref_off = 0;
    vrefcon_vref_on = 32;
    vrefcon_vref_alt = 64;

    accepted_value = 1;

    while(accepted_value)
        disp('#Valid vrefcon strings: vref off, vref on, vref alt');
        prompt = '#Enter vrefcon string: ';
        vrefcon = input(prompt,'s');
        if strcmp(vrefcon,'vref off')  || strcmp(vrefcon,'vref on')  ||
strcmp(vrefcon,'vref alt')
            accepted_value = 0;
        else
            disp('#Error. Unacceptable vrefcon string. Acceptable strings are: vref
off, vref on, vref alt.');
            disp('#Type "help vrefcon" for more details.');
        end
    end

    switch(vrefcon)
      case('vref off')
        vrefcon = vrefcon_vref_off;
      case('vref on')
        vrefcon = vrefcon_vref_on;
      case('vref alt')
        vrefcon = vrefcon_vref_alt;
    end

    message = [set_vrefcon, vrefcon];
    send_set_cmd(obj,set_vrefcon,message);
end
```

set_sysocal_enable.m

```
function set_sysocal_enable(obj)

    global set_sysocal_en;
    accepted_value = 1;
    while(accepted_value)
        prompt = '#Enable sysocal y/n? ';
        soc_en = input(prompt,'s');
        if strcmp(soc_en,'y') || strcmp(soc_en,'n')
            accepted_value = 0;
        else
            disp('#Error. Type either "y" or "n"');
        end
    end

    switch soc_en
        case 'y'
            soc_enable = 1;
        case 'n'
            soc_enable = 0;
```

```
    end

    message =  [set_sysocal_en, soc_enable];
    send_set_cmd(obj,set_sysocal_en,message);
end
```

set_delay_enable.m

```
function set_delay_enable(obj)

    global set_delay_en;
    accepted_value = 1;
    while(accepted_value)
        prompt = '#Enable delay y/n? ';
        d_en = input(prompt,'s');
        if strcmp(d_en,'y') || strcmp(d_en,'n')
            accepted_value = 0;
        else
            disp('#Error. Type either "y" or "n"');
        end
    end

    switch d_en
        case 'y'
            del_enable = 1;
        case 'n'
            del_enable = 0;
    end

    message =  [set_delay_en, del_enable];
    send_set_cmd(obj,set_delay_en,message);
end
```

set_rt_clock.m

```
function set_rt_clock(obj)

    % implement one of the time syncing protocols!

    global set_rtc;

    c = clock;
    c(1) = c(1) - 2000; % use only 18, 19 etc. and not 2018, 2019, etc.
    c(6) = floor(c(6)); % round seconds to the nearest integer
    c = uint8(c);
    % c(1) - year
    % c(2) - month
    % c(3) - day
    % c(4) - hour
    % c(5) - min
    % c(6) - sec

    message = cat(2,set_rtc,c); % concatenate command with clock
    send_set_cmd(obj,set_rtc,message);
end
```

set_keep_data_in_memory.m

```
function set_keep_data_in_memory (obj)

    global set_keep_data;
    accepted_value = 1;
    while(accepted_value)
        prompt = '#Keep data in memory y/n? ';
        kd = input(prompt,'s');
        if strcmp(kd,'y') || strcmp(kd,'n')
            accepted_value = 0;
        else
            disp('#Error. Type either "y" or "n"');
        end
    end

    switch kd
        case 'y'
            keep_data = 1;
        case 'n'
            keep_data = 0;
    end

    message =  [set_keep_data, keep_data];
    send_set_cmd(obj,set_keep_data,message);
end
```

set_cal_enable.m

```
function set_cal_enable(obj)

    global set_cal_en;
    accepted_value = 1;
    while(accepted_value)
        prompt = '#Enable calibration y/n? ';
        c_en = input(prompt,'s');
        if strcmp(c_en,'y') || strcmp(c_en,'n')
            accepted_value = 0;
        else
            disp('#Error. Type either "y" or "n"');
        end
    end

    switch c_en
        case 'y'
            cal_enable = 1;
        case 'n'
            cal_enable = 0;
    end

    message =  [set_cal_en, cal_enable];
    send_set_cmd(obj,set_cal_en,message);
end
```

set_operational_mode.m

```
function set_operational_mode(obj)

    global set_op_mode;
    global op_mode;
```

```
    accepted_value = 1;
    while(accepted_value)
        prompt = '#Enter operational mode: type either "rtc" or "bulk"\n#';
        som_en = input(prompt,'s');
        if strcmp(som_en,'rtc')
            op_mode = 0;
            accepted_value = 0;
        else
            if strcmp(som_en,'bulk')
                op_mode = 1;
                accepted_value = 0;
            else
            disp('#Error. Type either "rtc" or "bulk"');
            end
        end
    end

    message =  [set_op_mode, op_mode];
    send_set_cmd(obj,set_op_mode,message);
end
```

cli_help.m

```
function cli_help()
% Display available CLI commands
    disp('#');
    disp('#List of available commands:');
    disp('#get conv result');
    disp('#get data rate');
    disp('#get meas rate');
    disp('#get num bulk');
    disp('#get adc vrefcon');
    disp('#get sysocal en');
    disp('#get delay en');
    disp('#get dev state');
    disp('#get rtc');
    disp('#get ofc reg');
    disp('#get fsc reg');
    disp('#get keep data');
    disp('#get cal en');
    disp('#get op mode');
    disp('#set data rate');
    disp('#set meas rate');
    disp('#set meas start');
    disp('#set num bulk');
    disp('#set adc vrefcon');
    disp('#set sysocal en');
    disp('#set delay en');
    disp('#set dev state');
    disp('#set rtc');
    disp('#set time');
    disp('#set keep data');
    disp('#set cal en');
    disp('#set op mode');
    disp('#help');
    disp('#help vrefcon');
    disp('#help dev state');
    disp('#abort');
    disp('#exit');
    disp('#');
end
```

cli_help_vrefcon.m

```
function cli_help_vrefcon()
    disp('#Acceptable vrefcon strings are:');
    disp('#vref_off   Internal voltage reference is OFF');
    disp('#vref_on    Internal voltage reference is ON');
    disp('#vreff_alt  Int voltage reference is OFF when sleep, ON when converting');
    disp('#');
end
```